DISK ENCLOSED

Commodore Out — Amiga In p.2

AC's TECH For The Commodore AMIGA

Volume 4 Number 3
US $14.95 Canada $19.95

- •Amiga π
- •Assembly Programming for the Next Generation of Amiga Computers
- •True BASIC AmigaDOS Module
- •Huge Numbers Part 3
- •Re Color Revisited
- •A Simple AmigaDOS Handler
- •A Pair of Pickovers

03
0 74470 78640 5

Comeau C++ 3.0.1 With Templates Available. Affordable. Reliable. Supported.

Widely Available
Not only has Comeau C++ been ported to multiple operating systems and platforms, it is uncontestably the widest available commercial C++ anywhere. AmigaDOS, UNIX, MS-DOS, SunOS, RS/6000, SCO, SVR3, SVR4, XENIX and then some. You name it, we're there. And all with the same robust and up-to-date functionality.

Standard 3.0 ARM
Other C++'s claim conformance to AT&T USL's cfront, some still at the outdated cfront 2.0 or cfront 2.1 level. Comeau C++ was written as per the ARM and literally licensed from AT&T USL, so we don't need to claim anything. We literally are cfront. Further, we're 3.0.1, the latest release.

Latest Features
Templates, full nested types, multiple inheritance. They're all here. We are active and voting members of the ANSI C++ committee, so the latest working draft document changes will always be available as their specifications mature and become appropriate.

Excellent Reviews
Comeau C++ continues to get international acclaim in magazines such as *The C++ Journal*, *Dr. Dobb's Journal*, *Computer Language* and others. On the Amiga, recent coverage includes *AC's Tech* as well as *AmigaWorld* magazine. "netters" in comp.sys.amiga.reviews love it too.

Supports 3rd Party Libraries
We work with many of the 3rd party library vendors to ensure their libraries are consistent, reliable and language conforming. This means that most general purpose, niche specific, commercial and public domain libraries work unchanged with Comeau C++.

Amiga Support
Comeau C++ has been available on the Amiga for over two years. It is seriously supported and always kept up-to-date. Often it's even the first to get a feature. Comeau C++ is a translating compiler that generates C as its object code. On the Amiga, back end C compiler support includes SAS/C and Manx C. Dice C support is currently in beta. GCC support is currently in the works. Contact us for the latest news.

International Leadership
You don't need to wait weeks for your order. Comeau C++ is shipped internationally via UPS next day air. Add in a day or two for customs and it is there! Throw in a plethora of electronic mechanisms to contact us, and you might as well be next door.

Comeau Components
Imagine access to sets of class libraries that have had actual extensive use at the production level by the originators of C++ (AT&T, USL, etc.) since 1987. These reliable classes are what the Comeau Components are about. All designed and proven in the real world to have maximum run-time efficiency, reuse, productivity and functionality, allowing you to concentrate on your application's specifics. AmigaDOS specific additions will be available soon. Contact us for the latest news.

We Stand By What We Say
Far too many companies are market and sales driven. Comeau Computing is consumer driven. We'll stand by our product. Whether our free domestic shipping, much touted free technical support, 60-day offer or often free upgrades, we'll break the barriers. We're consumers too, and know you deserve better!

Easy Access
You'll always have fast and easy access to our sales, service and support staff. We can be conveniently accessed via many mechanisms, some accessible 24 hours/7 days a week, including: mail (91-34 120th Street, Richmond Hill, NY, 11418-3214), UseNet (acg@csanta.attmail.com), BIX (comeau), Compuserve (72331,3421), Prodigy (tshp50a), Voice (718-945-0009), Fax (718-441-2310).

Comeau
Computing
91-34 120th Street
Richmond Hill, NY 11418

Comeau C++.... Anything less is A-

AC's TECH AMIGA

ADMINISTRATION

Publisher:	Joyce Hicks
Assistant Publisher:	Robert J. Hicks
Administrative Asst.:	Donna Viveiros
Circulation Manager:	Doris Gamble
Asst. Circulation:	Traci Desmarais
Traffic Manager:	Robert Gamble
Marketing Manager:	Ernest P. Viveiros Sr.

EDITORIAL

Managing Editor:	Don Hicks
Editor:	Jeffrey Gamble.
Hardware Editor:	Ernest P. Viveiros Sr.
Video Consultant:	Oran Sands
Illustrator:	Brian Fox

ADVERTISING SALES

Advertising : Traci Desmarais
Donna Viveiros

1-508-678-4200
1-800-345-3360
FAX 1-508-675-6002

AC's TECH For The Commodore Amiga™ (ISSN 1053-7929) is published quarterly by PiM Publications, Inc., One Currant Road, P.O. Box 2140, Fall River, MA 02722-2140.

Subscriptions in the U.S., 4 issues for $44.95; in Canada & Mexico surface, $52.95; foreign surface for $56.95.

Second-class Postage paid at Fall River, MA 02722 and additional mailing offices.

POSTMASTER: Send address changes to PiM Publications Inc., P.O. Box 2140, Fall River, MA 02722-2140. Printed in the U.S.A. Copyright© 1994 by PiM Publications, Inc. All rights reserved.

First Class or Air Mail rates available upon request. PiM Publications, Inc. maintains the right to refuse any advertising.

PiM Publications Inc. is not obligated to return unsolicited materials. All requested returns must be received with a Self Addressed Stamped Mailer.

Send article submissions in both manuscript and disk format with your name, address, telephone, and Social Security Number on each to the Editor. Requests for Author's Guides should be directed to the address listed above.

AMIGA™ is a registered trademark of Commodore-Amiga, Inc.

Contents

Volume 4 Number 3

4 **Amiga π**
by Robert Davis

6 **Assembly Programming for the Next Generation of Amiga Computers**
by Christopher Jennings

13 **True BASIC AmigaDOS Module**
by T. Darrel Westbrook

22 **Huge Numbers Part 3**
by Michael Griebling

28 **Re Color Revisited**
by Dave Senger

48 **A Simple AmigaDOS Handler**
by Stephen Rondeau

53 **A Pair of Pickovers**
by Bill Nee

Departments

2 **Editorial**
32 **List of Advertisers**
33 **Source and Executables ON DISK!**

Startup Sequence

Commodore Out-Amiga In What Now?

Commodore Takes A Powder

Unless you have been trapped on another planet (or just not in daily contact with your favorite Bulletin Board System), you must already know that Commodore International has offered themselves to voluntary liquidation.

The end came suddenly for the Big C. On Friday, March 25, 1994, Commodore waited until after the New York Stock Exchange closed and then announced significant losses for the past quarter. Wall Street responded on March 28, 1994, when the stock exchange halted trading of Commodore stock.

Some industry observers believe Commodore's tactic was engineered to place pressure on their vendors and creditors. They needed more credit to produce enough products to generate greater sales. Commodore's biggest problem at the end of March was not that they were not selling machines, they just were not making enough machines to sell.

No matter how much overhead is cut, if a company cannot generate enough sales, they will not be profitable. In Commodore's case, the management had made significant cuts in overhead and payroll expenses worldwide. They had attempted to move several of their operations into less expensive quarters, and they had streamlined their distribution channels.

Apparently the creditors were not impressed. After a month, Commodore once again placed a late night call to the press and made the following statement:

"COMMODORE INTERNATIONAL LIMITED TO LIQUIDATE

New York, N.Y., April 29, 1994—Commodore International Limited (NYSE:CBU) announced today that its Board of Directors has authorized the transfer of its assets to trustees for the benefit of its creditors and has placed its major subsidiary, Commodore Electronics Limited, into voluntary liquidation. This is the initial phase of an orderly liquidation of both companies, which are incorporated in the Bahamas, by the Bahamas Supreme Court."

The reaction was almost immediate with Amiga users as they charged up their modems and e-mailed rumor after rumor. Sony, Philips, Samsung, and other leading electronic companies were offered up one by one and then in groups as the company or group about to announce that they had purchased the Amiga technology.

Commodore has made no further official or unofficial comments concerning the liquidation. Their official spokesperson apparently is not accepting or returning phone calls. The Bahamas organization who is handling the matter has no further word to offer. This has left the Amiga market open to rumor and innuendo, but for most of us in the Amiga market, we see no real difference.

Amiga Support

One of the best reactions to come from this situation has been the continued support and public declarations by many Amiga vendors. From Amiga developers such as Blue Ribbon to distributors such as CEI, companies have been offering their customers strong assurances that they would continue to develop for and support the Amiga.

One noticeable comment came from IAM's president, Dale L. Larson, who stated, "The Amiga is important. Commodore is irrelevant." Mr. Larson was quoted in a press release generated by IAM. He went on to say, " Even if Commodore disappears and no one else picks up production of the Amiga, there is a large existing base of machines which will remain valuable for many years. We intend to help current Amiga users to get the most from those systems. Further, we believe that licensing agreements or other arrangements will likely allow production of new machines by someone."

Mr. Larson's sentiments have been repeated by the majority of people in the Amiga market. From vendors, users, dealers, and distributors, the feeling remains the same, Commodore was not helping the Amiga, so anyone who comes in now will probably do a better job.

New Technology

It is virtually assured that some company will continue the Amiga line, but one nagging question is what will happen to future development on the Amiga?

The work on the AAA chip set was last reported to be only one silicon test away from completion. From reports by departing engineers, Commodore has removed almost all of the development staff. However, they have left in place a dedicated minority who know enough about the entire Amiga technology to begin the process again with a new crew. When the Amiga is sold, the new owner will not only have a proven platform in graphics, video, entertainment and more, but they will also have the next level of this technology near completion. Some insiders have stated that the new levels of Amigas could be in the hands of the dealers in as little as six months.

"The Amiga is important. Commodore is irrelevant."

Dale L. Larson, President IAM

Change is Opportunity

Every self motivation or business book I have ever read, always stresses the opportunity that change brings. With a new company at the helm of the Amiga, we may see the market share and penetration we have expected for so long.

It is even possible that a new company will bring along its own ideas for improving the Amiga market. Wouldn't it be interesting if these included business applications, advanced graphic software by companies from other platforms, increased educational presence, and more.

A new company will generate a new respect for the Amiga. Remember, Commodore did not create the Amiga, they bought it. Isn't it interesting that the Amiga will outlive Commodore and give someone else a turn.

Sincerely,

[illegible]

Don Hicks
Managing Editor

Digital Imagery

•35MM Slides
•35MM Negatives
•Color Prints
•Color Transparencies

•Image Modification
Color Correction
Format Conversion

•Design Service
Slide Presentations
Product Packaging

• Prepress Services

For additional information please phone or write to:

Digital Imagery
945 Walnut Street
Fall River, MA 02720

(Voice) 508-676-0844
(Fax) 508-676-5186
(BBS) 508-676-9308

Technical Writers
Hardware Technicians
Programmers
Amiga Enthusiasts

Do you work your Amiga to its limits? Do you do create your own programs and utilities? Are you a master of any of the programming languages available for the Amiga? Do you often find yourself reworking a piece of hardware or software to your own specifications?

If you answered yes to any of those questions, then you belong writing for *AC's TECH*!

AC's TECH for the Commodore Amiga is the only Amiga-based technical magazine available! We are constantly looking for new authors and fresh ideas to complement the magazine as it grows in a rapidly expanding technical market.

Share your ideas, your knowledge, and your creations with the rest of the Amiga technical community—become an *AC's TECH* author.

For more information, call or write:
AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

1-800-345-3360

AMIGA π

by Robert Davis

Recycling is a Good Thing. Junk cars, aluminum cans, and newspapers are recyclable, and so are old computer programs. The program described in this article originally appeared as an Apple II Integer Basic listing in the magazine *Micro* in the late 1970s. Here it is, recycled as an Amiga program written in AMOS Basic.

News stories about super-computers calculating PI to millions of digits appear regularly. The old magazine article mentioned above listed a fairly short program which would calculate PI to a thousand digits in a computer with only 16K of RAM. An Amiga, with 512K or more memory, does quite well, especially when compared to the computers of fifteen years ago. The original author of the program, Robert Bishop, wrote that he took the formula for calculating PI from the 1940 edition of <u>*Mathematics and [the] Imagination*</u>, by Edward Kasner and James Newman. The book is still available, republished in 1989 by Microsoft Press. The ISBN number for the paperback edition is 1-55615-104-7, and it only costs $8.95 for a fascinating read.

The formula used by this program for determining the value of PI is one of several converging series which all give the same result. From page 77 of the 1989 edition, and slightly rearranged, we find that:

```
        PI = 16*(1/5 - 1/3*5^3 + 1/5*5^5 - 1/7*5^7 + 1/9*5^9 - ....) -
4*(1/239 - 1/3*239^3 + 1/5*239^5 - 1/7*239^7 + 1/9*239^9 - ....)
```

And that is what this program does. The variable names should offer enough explanation of the DIVIDE, ADD, and SUBTRACT routines. The program multiplies by doing repeated additions. Nothing in the program is specific to any model of Amiga. I found it quite easy to modify the Apple II Integer Basic program into AMOS (1.36). Only one function did not work as expected. The original program used exponentiation at one point. PASS^2 should work, but AMOS (1.36) gave wrong results with that code. PASS*PASS does give correct results. Speed comparisons are unfair, but inevitable. Robert Bishop reported that his Apple II took 40 hours to calculate PI to 1000 places. The compiled AMOS program does the same thing in about nineteen minutes on my stock Amiga 500, and about three(!) minutes on my 25 megahertz Amiga 3000.

Since incomplete calculations give incorrect results, the last one or two digits of PI as computed by this program are often wrong. To get the correct result for a thousand digits, you must specify 1001. Experiment seems to show that adding 2 to the number of digits you really want to see will give an accurate calculation of PI.

Does anyone need PI to hundreds or thousands of digits precision? Nope. I have read that NASA, in its computer programs for the Space Shuttle, uses a value of 3.14. But the program to solve the equation is interesting, parts may be useful in other programs, and the program output serves quite well as a source of pseudo-random numbers. Finally, I have always wanted to know the value of PI to thousands of digits. And now I do.

Listing

```
Rem the AMIGA_PI program in AMOS ... 15 March 1994
Rem by Robert Davis at the Boarding House BBS (913) 827-0744
Rem the original idea for this program comes from
Rem "The Best of MICRO" Volume 1, pages 85-86, by Robert Bishop
Set Buffer 8
Screen Open 4,640,200,4,Hires
Palette 0,$177,$F0,$FFF : Rem R G B values (0 1 2 3)
Rem colour 1 = background, colour 2 = text
Erase 10 : Erase 11 : Erase 12 : Rem clear memory banks
Dim CONSTANT(2) : CONSTANT(1)=25 : CONSTANT(2)=239
TEN=10 : LINE=2 : CULUMN=4 : Curs Off
Locate CULUMN,LINE
Print "A PI calculation exercise in AMOS Basic." : Inc LINE : Inc LINE
Locate CULUMN,LINE
Print "Use right mouse button to pull down the menu." : Inc LINE
```

```
Rem the menu structure
Menu$(1)=" AMIGA_PI     "
Menu$(1,1)=" About ...          "
Menu$(1,2)=" Output to screen  "
Menu$(1,3)=" Output to printer "
Menu$(1,4)=" Quit              "
MENSTRT:
Wind Save : Rem remember what the window covers
Menu On
Do
  If Choice Then Exit
Loop
Menu Off
On Choice(2) Goto ABTPGM,SCRN,PRTR,PGMXIT
Stop : Rem error trap
ABTPGM:
Wind Open 5,16,32,72,10,1
Curs Off
Print " Original program for Apple II by Robert Bishop"
Print " Published in 'Best of Micro', volume 1, pp 85-86"
Print
Print " Amiga adaptation by Robert Davis, Salina, KS."
Print " rdavis@nyx.cs.du.edu  or  The Boarding House BBS 913 827-0744"
Print
Print " Press a key to continue."
Wait Key
Wind Close
Goto MENSTRT : Rem loop until user makes choice
PRTR: OWTPRTR=1 : Goto HWMANY
SCRN: OWTPRTR=0
HWMANY:
Inc LINE : Locate CULUMN,LINE : Curs On
Print "To how many digits shall we calculate PI";
Input SIZE : Curs Off
Inc LINE : Locate CULUMN,LINE
Print "Working, please wait."
On Error Goto GENERICERR
MEMREQUIRED=4*SIZE
Reserve As Data 10,MEMREQUIRED : Rem save some space in RAM
Reserve As Data 11,MEMREQUIRED
Reserve As Data 12,MEMREQUIRED
Rem error checks should go here
POWER=Start(10) : TERM=Start(11) : RESULT=Start(12) : Rem the memory banks
names
Rem program main loop
 For PASS=1 To 2
    Gosub INIT
INLP: Rem interior loop
    Gosub COPEE
    POYNT=TERM : DIVIDE=XPN : Gosub DIV
    If SIGN>0 Then Gosub ADX
    If SIGN<0 Then Gosub SBX
    XPN=XPN+2 : SIGN=-SIGN
    POYNT=POWER : DIVIDE=CONSTANT(PASS) : Gosub DIV
    If PASS=2 Then Gosub DIV
    If ZERO<>0 Then Goto INLP
 Next PASS
Rem output the result
If OWTPRTR=1 Then Goto DUPRTR
Inc LINE
Locate CULUMN,LINE
Print "The value of PI to";(TEN/100+1)*SIZE;" decimal places ..." : Print
Print Peek(RESULT);".";
For PLACE=RESULT+1 To RESULT+SIZE
  Print Right$(Str$(Peek(PLACE)),1);
Next PLACE
Print : Print : Print "    The last one or two digits are always suspect."
Goto PGMFIN
DUPRTR: Rem do printer
PRCHR=2 : Rem we print '3.' before starting the count
Lprint " The value of PI to";(TEN/100+1)*SIZE;" decimal places." : Lprint
Lprint Right$(Str$(Peek(RESULT)),1);".";
For PLACE=RESULT+1 To RESULT+SIZE
  Lprint Right$(Str$(Peek(PLACE)),1); : Inc PRCHR
  If PRCHR=76
    Lprint
    PRCHR=0
  End If
Next PLACE
Lprint : Lprint " The last one or two digits are always suspect."
Lprint Chr$(12);
PGMFIN: Rem the program is finished now
Print : Print "    Press a key to exit."
Wait Key
Stop : Rem internal end of program
```

```
Rem the DIV divison routine
DIV:
DIGIT=0 : ZERO=0
For PLACE=POYNT To POYNT+SIZE
  DIGIT=DIGIT+Peek(PLACE)
  QUOTIENT=DIGIT/DIVIDE
  RESIDUE=DIGIT mod DIVIDE
  ZERO=ZERO or(QUOTIENT+RESIDUE)
  Poke PLACE,QUOTIENT
  DIGIT=TEN*RESIDUE
Next PLACE
Return
Rem the ADX addition subroutine
ADX:
CARRY=0
For PLACE=SIZE To 0 Step -1
  SUM=Peek(RESULT+PLACE)+Peek(TERM+PLACE)+CARRY
  CARRY=0
  If SUM<TEN Then Goto OWTADX
  SUM=SUM-TEN
  CARRY=1
OWTADX:
  Poke RESULT+PLACE,SUM
Next PLACE
Return
Rem the SBX subtraction routine
SBX:
LOAN=0
For PLACE=SIZE To 0 Step -1
  DIFFERENCE=Peek(RESULT+PLACE)-Peek(TERM+PLACE)-LOAN
  LOAN=0
  If DIFFERENCE=>0 Then Goto OWTSBX
  DIFFERENCE=DIFFERENCE+TEN
  LOAN=1
OWTSBX:
  Poke RESULT+PLACE,DIFFERENCE
Next PLACE
Return
INIT: Rem initialize memory banks and variables
  For PLACE=0 To SIZE
    Poke POWER+PLACE,0
    Poke TERM+PLACE,0
    If PASS=1
       Poke RESULT+PLACE,0
    End If
  Next PLACE
    Poke POWER,16/(PASS*PASS)
    If PASS=1
       DIVIDE=5
    Else
       DIVIDE=239
    End If
    POYNT=POWER : Gosub DIV
    XPN=1 : SIGN=3-2*PASS
Return
Rem the COPY routine .. copy 'power' into 'term'
COPEE:
For PLACE=0 To SIZE
Poke TERM+PLACE,Peek(POWER+PLACE)
Next PLACE
Return
GENERICERR:
Print : Print " Generic error message."
Print " Press a key to exit."
Wait Key
PGMXIT:
End
```

✔

Please Write to:
Robert Davis
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

Assembly Programming for the Next Generation of Amiga Computers

by Christopher Jennings

The introduction of the AA chip set machines created the first major gap in Amiga compatibility. With this new chip set Commodore has radically reworked the foundation of the machine in almost every conceivable way. Wisely, they also took this opportunity to update the aging 68000 CPU; the 68020 is now the low-end processor of the new Amiga generation. Although many users have already been taking advantage of the increased speeds of faster processors, very few applications have been written to take advantage of the changes to the processor itself.

The advent of the AA machines finally presents the new starting point that Amiga developers have been looking for since 2.0 and the ECS chip set were released. Choosing to develop a project for AA machines only, or offering two different versions affords programmers a threefold benefit; first the availability of the improved AA graphics and other capabilities (and the chance to exploit a new market which is still underdeveloped at this point), second the ability to use all the powerful new features added in 2.0 and following operating systems without limitations in the interest of backward compatibility, and third the ability to take advantage of a powerful new processor.

This article concentrates on some of the many advanced features of the 68020 and higher processors, and is designed for experienced and learning 68000 programmers to expand their existing knowledge base. Those unfamiliar with Assembly Language programming on the Amiga are advised to work with a more introductory text (such as the ongoing series appearing in this magazine or any of several books on the subject) before proceeding.

Many application writers will be unwilling to venture into requiring AA machines for their software until a larger user base forms; on the other hand AA-only games are already being developed on a large scale. Certainly if you are programming for the new CD^{32} machine you should take full advantage of the situation. Furthermore it is generally games that are written entirely in Assembly Language in the first place. Given this situation, this article does lean slightly into the games programming area. At the same time, the information contained herein need not be limited to that application. Much as with the introduction of 2.0, 3.0, and now 3.1, programmers may wish to take advantage of features that are available by including separate bits of code for higher processors. For example, suppose you are writing a program in C that utilizes a sort routine. Since this function is critical to the program, you may wish to write that part of the code in Assembly Language. Furthermore, you might write two versions of the routine, one for the 68000 and 68010, and one for the 68020 and higher. This guide is also valuable for this purpose. Although this one article cannot teach all of the new instructions and features available with the 68020, it does contain a brief overview of these things, and a guide to utilizing them effectively. Its primary purpose is to give insight to experienced 68000 programmers to assist their efficient use of the new chips.

Processor/AA Detection

You can determine the nature of the system you are running on easily by reading the bits of ExecBase->AttnFlags (whose offset from ExecBase is $128). Bit 1 of this word will be set if the processor is a 68020 or higher (see the detailed explanation below for more information). However, in the event you are planning on carrying out low level coding (such as writing your own operating system based on the Amiga hardware) or you happen to be programming a non-AmigaDOS machine in the future, things get more complicated. Normally, CPU/FPU/MMU detection involves a series of fairly tricky steps which will execute only in supervisor mode. However, if you are only interested in whether or not a computer has an '020 or higher processor, you can take advantage of an oversight in the design of the 68000 and 68010 chips. The following lines return a 1 (True) if the processor is at least an '020, or a 0 (False) otherwise, in d0:

Example 1: System Independent 68020 Detection

```
check_020:
   moveq #1,d0
   lea      chk_table,a0
   move.b   0(a0,d0.L*4),d0
   rts
chk_table:
   dc.l     $0,$1000000
```

The routine works this way: in the 68000/68010 chips, the extension word form of indexed addressing modes is not checked. Therefore, the correct offset will be computed for an '020 or above (and point to the 1 byte, of course), but the lower chips will ignore the scaling, yielding a 0. Assembling the code without error may require specifying a higher chip (i.e. the '020) as an option; see your assembler manual for details. Of course, the standard OS method of processor detection is to use the following bit of code:

Example 2: System Flag Processor Detection

```
AttnFlags equ $128
   move.l  4.w,a6              ; Find ExecBase
   move.w AttnFlags(a6),d0     ; Get CPU/MMU/FPU data
```

Then d0's bits will be set according to the following table:

Table 1: Processor Bits In AttnFlags

Bit	Meaning (if set)	Comment
0	MC68010 or higher processor	
1	MC68020 or higher processor	
2	MC68030 or higher processor	Requires V37+ Exec
3	MC68040 or higher processor	Requires V37+ Exec
4	MC68881 (or MC68882) FPU	
5	MC68882 FPU	Requires V37+ Exec
6	MC68040 FPU	Requires V37+ Exec; if bit 6 is set but 4 and 5 are not, the math emulation code has not been loaded.

Note that higher processors also set all bits below them, so on an '030, bits 0 to 2 will be set. Bit 6 (MC68040 FPU) should be ignored if bit 3 (MC68040 CPU) is reset (zero). These bits are defined and explained completely in execbase.i.

Similar to processor detection, the system provides a way to read graphics chip availability (requiring a library version of 39 or higher will not work, since Workbench 3.1 will run just fine without an AA chip set). This is in GfxBase->gb_ChipRevBits0 (offset $EC); the bit to check is 2, for the presence of the Alice chip. The following lines will demonstrate the technique; it is assumed that you have opened graphics.library and placed its base in a6 prior to executing this code:

Example 3: Chipset Detection

```
gb_ChipRevBits0 equ $ec                        ; GfxBase offset
GFXB_AA_ALICE equ $2                           ; Alice bit
   btst  #GFXB_AA_ALICE,gb_ChipRevBits0(a6)  ; Perform test
   beq.s  No_AA_Chipset                       ; Act on result
```

Some publications may advocate hardware-based methods of determining the presence of an AA chipset. However, the bits involved will almost certainly change in the future, and they should not be used. Note, however, that the SetPatch command must be executed under 3.0 (v39) and 3.1 (v40) before this routine is used if it is to return the correct result. In fact, it is the SetPatch command that actually activates the AA chips. You can get around this requirement by doing the same thing that SetPatch does on start-up, which is to call the SetChipRev() function in the graphics library. The function is called with the desired chipset as its argument; currently this may consist of the following:

Table 2: SetChipRev() Arguments

Value	Chipset Requested
$00000000	Original chipset
$00000003	Enhanced chipset (ECS)
$0000000f	Advanced Graphic Architecture (AGA/AA)
$ffffffff	Best available

This function is only available in V39 and above; if you don't have at least Kickstart 3.0, then you don't have an AA machine. On the other hand, having Kickstart 3.0 does not mean you must have an AA chipset, and it should not be used as a test. The function returns the state of the chipset on exit, so you should check the result to ensure you actually got what you asked for. This function should generally only be called once. If you are writing code that requires AA, you might want to do the following:

1 Open graphics.library with a minimum version number of 39
2 If this fails, AA graphics are unavailable; issue your error message
3 Check the GfxBase chipset revision bits (see Example 3)
4 If this is successful, AA graphics are available; continue your program
5 If this fails, AA chips may still be present. If you are using system routines for your graphics code (and you should be whenever possible), then call SetChipRev($ffffffff). If you use the hardware, and you use any AA-only registers or bits, your code will not work on the AAA and later chipsets, so you must call SetChipRev($f) to request AA specifically. Original/ECS register calls should still work (thus allowing you to set up 12-bit copper list rainbows and other things not yet possible with the OS).
6 Check the results to see if your requested chips are there.

If you are writing software in both Original/Enhanced and AA versions, and you want to include things like 24-bit color custom copper lists for your backgrounds, you might want to consider writing three versions, as follows:

1 Original/Enhanced: Use OS calls when possible, use hardware registers only when calls are not available in Kickstart 1.2.
2 AA Only: Use OS calls when possible, use AA registers for 24-bit copper lists and things not implemented in the OS.
3 AAA+: Use 3.1 OS calls; fall back to original chipset when using hardware registers. It is clear that these registers

will exist in the AAA chipset (and that AA registers will not). Beyond AAA, their future is uncertain. This method will probably produce AAA compatible software, but there is no guarantee.

The 68020 Processor

With the release of the 68020 in 1985 came many major enhancements over the older 68000 and 68010, the most significant of which was probably the switch from a 16-bit data path and 24-bit address space to full 32-bit data and addresses. The '020 was also the first chip in the family to have an instruction cache (later processors also included a data cache and greatly increased the sizes of the caches). This was a major development over the 68010's loop mode by holding 256 bytes of the most recently used instruction stream within the processor. Therefore, loops and branches which jump back within that space will execute significantly faster than on older processors. On machines without special graphics support chips (such as a Macintosh), the cache gives these computers enough speed to support the demands of a graphical user interface. The '020 also introduced some new addressing modes designed primarily for faster array access and a new data type, the bitfield, which is invaluable for arbitrary bit boundary accesses commonly found in graphics applications and certain compression schemes. A number of new addressing modes and instructions which were also added are outlined below.

Addressing Modes

The 68020 processor supports six new addressing modes (for a total of eighteen): (1) Address Register Indirect With Index and Base Displacement, (2) Memory Indirect Post-Indexed, (3) Memory Indirect Pre-Indexed, (4) Program Counter Indirect With Index and Base Displacement, (5) Program Counter Memory Indirect Post-Indexed, (6) Program Counter Memory Indirect Pre-Indexed. In standard Motorola syntax, these are written as:

Table 3: New 68020+ Addressing Modes

Nº	Syntax	Example
1	(bd,Ax/SP,Rx.SIZE*SCALE)	(128,A0,D0.L*4)
2	([bd,Ax/SP],Rx.SIZE*SCALE,od)	([128,A2],D0.W*8,32)
3	([bd,Ax/SP,Rx.SIZE*SCALE],od)	([16,SP,D7.W*2],4)
4	bd(Ax/SP,Rx.SIZE*SCALE)	32(A0,D1.W*2)
5	([bd,PC],Rx.SIZE*SCALE,od)	([1024,PC],D3.L*4,2)
6	([bd,PC,Rx.SIZE*SCALE],od)	([2,PC,D3.L*4],4)

Where bd is a 16/32-bit base displacement, Ax/SP is an address register or the stack pointer, Rx.SIZE*SCALE is an index register (Rx) of 16/32-bit size (.SIZE) multiplied by a given scale (*SCALE). SCALE can be any of 1, 2, 4, or 8, for easy access to byte, word, longword and quadword values. The od is an outer displacement added to the calculated address. The use of a base register of ZA0-ZA7 instead of the usual Ax/SP base register will create a zero suppressed address register setting that means that the register will not be used in the calculation. If the index register is replaced with a zero, it is replaced by zero in the address calculation.

Address Register Indirect With Index and Base Displacement is similar to the familiar Address Register Indirect With Index and 8-bit Displacement, except that it uses a scaled index register of 16 or 32 bits. Also, as explained above, the index register can be scaled by byte, word, longword and quadword sizes for easy array access. All of the elements are optional, and if all are omitted, the effective address is zero. Further, since the address register is optional, this can actually be used to create a "data register indirect" mode, called base displacement mode. The assembler is responsible for deciding when to use the 8, 16, or 32-bit displacement modes. The calculation that occurs is as follows:

<effective address> = bd+Ax+(Rx.Size*Scale)

Memory Indirect Post-Indexed uses a pointer calculated using the address register and base displacement. The contents of the longword pointed to, plus the optional scaled index and/or 32-bit constant (the outer displacement) generate the address. The address calculation is:

<effective address> = Value at (bd+Ax) + Rx*Scale+od

Memory Indirect Pre-Indexed is identical to the post-indexed form, except that the index register is used in the pointer (indirect) address calculation instead of the operand address calculation. In this mode, the effective address is calculated as:

<effective address> = Value at (bd+Ax+Rx*Scale) + od

Program Counter Indirect With Index and Base Displacement uses the program counter and displacement added to the value of the optional scaled index register. The address is computed as:

<effective address> = bd+PC+(Rx.Size*Scale)

Program Counter Memory Indirect Post-Indexed is the PC-relative form of the memory indirect post-index mode, and simply uses the PC instead of an address register (see above). The effective address is:

<effective address> = Value at (bd+PC) + Rx*Scale+od

Program Counter Memory Indirect Pre-Indexed is a PC-relative memory indirect pre-index using the PC instead of an address register (see above). The address is computed using:

<effective address> = Value at (bd+Ax+Rx*Scale) + od

There is also an addition to an old addressing mode: the 68000/68010 Address Register Indirect With 8-bit Displacement can now also be used with index scaling; this scaling is ignored on the older processors. Thus it now takes the form:

```
operand (od,Ax/SP,Rx.SIZE*SCALE)
```

Instruction Set Synopsis

The following table is a brief overview of the new instructions available on 68020 and higher processors; for a more detailed explanation, see a reference manual on the processor in question. The most useful instructions for most programmers will probably be the bitfield instructions and the 32-bit multiply and divide instructions.

Table 4: New 68020+ Instructions

Opcode	General Syntax	Description
BFCHG	<ea>{offset:width}	Test and change a bitfield
BFCLR	<ea>{offset:width}	Clear a bitfield
BFEXTS	<ea>{offset:width},Dx	Bitfield extract signed
BFEXTU	<ea>{offset:width},Dx	Bitfield extract unsigned
BFFFO	<ea>{offset:width},Dx	Bitfield find first one
BFINS	Dx,<ea>{offset:width}	Bitfield insert
BFSET	<ea>{offset:width}	Bitfield set
BFTST	<ea>{offset:width}	Bitfield test
BKPT	#<data>	Send breakpoint acknowledge cycle
CALLM	#<data>,<ea>	Call module (1)
CAS	Dx,Dy,<ea>	Compare and swap (3)
CAS2	Da:Db,Dx:Dy,(Rm):(Rn)	Compare and swap two operands (3)
CHK2	<ea>,Rx	Check register against 2 bounds
CMP2	<ea>,Rx	Compare register against 2 bounds
cpBcc	<label>	Branch on coprocessor condition
cpBcc	<label>	Branch on coprocessor condition
pDBcc	Dn,<label>	Decrement and branch on coprocessor condition
cpGEN	<parameters>	Pass command to coprocessor
cpRESTORE	<ea>	Restore coprocessor state
cpSAVE	<ea>	Save coprocessor state
cpScc	<ea>	Set on coprocessor condition
cpTRAPcc	#<data>	Trap on coprocessor condition
DIVS.L	<ea>,[Dh:]Dl	32 to 32/64-bit signed division
DIVU.L	<ea>,[Dh:]Dl	32 to 32/64-bit unsigned division
EXTB	Dx	Sign extend a byte
MOVE	CCR,<ea>	Move condition code register (2)
MOVE	<ea>,CCR	Move to condition code register (2)
MOVEC	Rx,Ry	Move control register (2)
MOVES	Rx,<ea>/<ea>,Rx	Move address space (2)
MULS.L	<ea>,[Dh:]Dl	32 to 32/64-bit signed multiply
MULU.L	<ea>,[Dh:]Dl	32 to 32/64-bit unsigned multiply
PACK	-(Rx),-(Ry),#<data>	Pack two bytes into BCD format
RTD	#<data>	Return and deallocate (2)
RTM	Rn	Return from module (1)
TRAPcc	#<data>	Trap on condition code
UNPK	-(Rx),-(Ry),#<data>	Unpack BCD to two bytes

Notes:

1 This instruction is only available on the 68020 and should not be used under normal circumstances.
2 This instruction is also available on the 68010.
3 This instruction uses a read/modify/write cycle and should not be used under normal circumstances.

Bitfields

A bitfield is a newly supported datatype that consists of a series of bits, starting with any bit in a byte and continuing from 1 to 32 bits. Bitfields are specified using an offset and a width. Consider the following instruction:

```
BFSET (A0){8:4} ; Set third byte after (A0)
```

This instruction will affect the longword pointed to by A0, starting at the 8th bit (counting from 0, of course) from the left (MSB) and continuing 4 bits. Notice that the bit-counting is little endian, while the 680x0 is normally big endian in nature. That is, {8:4} does not start at bit 8 of the longword pointed to by A0, it starts at bit 31-8 (23). Similarly, it does not then continue from bit 23 right to bit 26, but instead proceeds left to bit 20. The single operand bitfield instructions are as follows:

BFTST <ea>{offset:width}	Test bitfield
BFCLR <ea>{offset:width}	Test bitfield and clear it
BFSET <ea>{offset:width}	Test bitfield and set it
BFCHG <ea>{offset:width}	Test bitfield and invert it

These instructions set condition codes as follows:

Code	Effect
N	Set if the most significant bit was set
Z	Set if all bits are clear
C	Cleared
V	Cleared
X	Unaffected

The BFSET, BFCHG, and BFCLR instructions can set, reverse (compliment), or clear bits, which can be useful for various drawing and filling operations. The two operand bitfield instructions are as follows:

BFEXTU <ea>{offset:width},Dn	Extract a bitfield
BFEXTS <ea>{offset:width},Dn	Extract and sign extend
BFINS Dn,<ea>{offset:width}	Insert a bitfield
BFFFO <ea>{offset:width},Dn	Find first set bit

These instructions result in the following condition code state:

Code	Effect
N	Set if the most significant bit is set
Z	Set if all bits are clear
C	Cleared
V	Cleared
X	Unaffected

The BFEXTU and BFEXTS instructions are used to extract (and if desired, sign extend) bitfields from any arbitrary boundary. This can be useful to speed intense operations on a given bitfield, as well as expand the range of operations available to act on them. They can be particularly helpful for writing compression/decompression and floating point routines.

BFEXTU/BFEXTS These instructions extract a bitfield from the source operand, right-justify it, sign extend it in the case of BFEXTS, and place it in the destination register.

➛

BFINS This instruction extracts the <width> lower bits of the source data register, and inserts them into the destination bitfield.

BFFFO The bit offset of the most significant set bit in the bitfield is stored in the data register. If the entire bitfield is clear, the sum of the offset and width is stored in the destination.

Bit Shift Instructions

Because the 68020 includes a barrel shifter, all shift instructions require the same number of cycles to execute no matter how many bits are involved. Different types of shift instructions still require different execution times (for example, ASL/ASR instructions are still slower than LSL/LSRs). This means that certain optimization techniques, specifically the replacement of LSLs and ASLs with various numbers of ADD Dn,Dn instructions are no longer as effective. The break even point for LSL is two shifts. Similar results occur with ROXL and ADDX Dn,Dn.

Other Instructions

CMPI/TST These instructions now support PC-relative mode.

CMP2 This instruction is an extension of the CMP operands and is used to compare a register against two bounds. The effective address is a pointer to the lower and upper bounds, which are the same size as the instruction. If the register is outside the bounds, the C flag is set. If it is on either boundary, Z is set, and if it is within the bounds, both are clear. If the register is an address register, byte and word data will be sign extended prior to comparison.

EXTB This new instruction sign extends a byte to a longword and is faster than the EXT.W EXT.L sequence required for 68000's.

MOVE16 This instruction is only available on the 68040 processor; it uses the burst mode to rapidly move data. The lower 4 bits of addresses are masked off to align them on 16-byte boundaries. A linear block of 16 bytes is then copied from the source to the destination. Obviously, care must be taken if this instruction is to be used in Chip memory since it uses the burst mode.

MULS/MULU/DIVS/DIVU These instructions are now capable of 32-bit x 32-bit to 32-bit, 32-bit x 32-bit to 64-bit multiplication and 32-bit / 32-bit to 32-bit and 64-bit / 32-bit to 32-bit division (with 32-bit modulos). The term used for the 64-bit datatype is a "quadword" (.q), and it is specified by combining two 32-bit registers into a high and low 64-bit unit (Dh:Dl). Of course, these instructions are tremendously faster than the algorithms required for the 68000.

PACK/UNPK These instructions compliment the previous instructions for dealing with Binary Coded Decimal number strings (ABCD, NBCD, and SBCD). The PACK instruction takes two bytes and changes them to high nybble/low nybble form. The UNPK instruction expands these values back to their original form. The displacement value is subtracted or added from the bytes to convert ASCII or other code methods to plain values from zero to nine when packing, or to change the BCD to other formats on unpacking.

RTD This instruction is an extended form of the RTS instruction which should be particularly useful for compiler writers. It pops the PC off the stack, then adds a 16-bit displacement to the SP. This makes it simple to clear parameters pushed onto the stack by a calling routine. For example, suppose a program module is called with PARAM bytes of parameters on the stack, and the routine requires LOCAL bytes of local data space. The following examples represent 68000 and 68010+ versions of this situation:

Example 4: Use of the RTD Instruction

```
;
; 4a) 68000 code
;

Procedure1:
   link     a5,#-LOCAL          ; Allocate local data space
   movem.l  d2-d7/a2-a6,-(sp)   ; Save registers

   ; Perform procedure code

   movem.l  (sp)+,d2-d7/a2-a6   ; Restore registers
   unlk     a5                  ; Deallocate local data
   move.l   (sp),(PARAM,sp)     ; Deallocate parameters
   lea      (PARAM,sp),sp
   rts                          ; Return

;
; 4b) 68010+ code
;

Procedure2:
   link     a5,#-LOCAL          ; Allocate local data space
   movem.l  d2-d7/a2-a6,-(sp)   ; Save registers

   ; Perform procedure code

   movem.l  (sp)+,d2-d7/a2-a6   ; Restore registers
   unlk     a5                  ; Deallocate local data
   rtd      #PARAM              ; Deallocate parameters and return
```

Performance Optimization

There are various programming techniques that you should use wherever possible in order to maximize processing speed. The '020 has many enhancements to boost its speed, but they cannot be used to full effect without careful programming. Perhaps most importantly, all loops should be kept under 256 bytes when feasible; this will allow them to be executed completely from the on-chip instruction cache. The cache also means that self-modifying code (rarely used anymore) is out, since instructions that decode themselves within the cache (which is larger on newer chips like the '040 and '060) will only be modified in actual memory, and thus not be executed.

Another consideration is the 32-bit nature of the chip. Even though the machine no longer crashes when addresses of words and longwords are not aligned properly, performance is degraded. Since all of the 68020's memory accesses are on longword boundaries, accessing an address on, for example, a word boundary means that the processor will read in two longwords: one containing the first half of the address, one containing the second. Therefore, longword accesses on longword bounds are twice as fast. The stack should always be maintained on a longword boundary, and aligning the entry points of routines in your program will also speed things up. These alignments may add a few pad bytes to the code, but they can significantly increase speed.

You should also be aware of the pipelined nature of the Motorola processors. This means that the processor can perform certain parts of instruction execution in parallel. As a simplified example, your code might consist of three instructions: A, B, and C. First, instruction A is fetched. Then, while A is being decoded, instruction B is fetched. At the third stage, A is executed, B is decoded, and

C is fetched, and so on. This pipeline can only work efficiently if it is kept open by using instructions in certain sequences to avoid problems. You should try to arrange your code so that memory accesses do not occur immediately after each other; the same stalling is caused by using an address register that is updated on one line, and accessed on the next. Keep in mind that write accesses to Chip RAM generally incur wait-states. For example:

Example 5: Avoiding Wait-States In Execution

```
;
; 5a) Poorly Organized Code
;
    move d0,(a1)+  ; Save variable A
    move d1,(a1)+  ; Save variable B
    add   d2,d0 ; A'=A+C
    sub   d2,d1 ; B'=B-C

;
; 5b) Faster Equivalent Code
;
    move  d0,(a1)+ ; Save variable A
    add   d2,d0 ; A'=A+C
    move  d1,(a1)+ ; Save variable B
    sub   d2,d1 ; B'=B-C
```

The base AA systems, the A1200 and the CD^{32} game machine, sport 2 megs of Chip RAM and no Fast RAM in their base configurations. The Chip RAM tends to have a large number of wait-states, while the system ROM has none. This is a good argument for using the system routines instead of coding your own whenever possible, since the ROM code will generally execute faster. For the same reasons, it is sometimes better to use calculations for values that may have been faster stored in a table before, given the increased speed of the '020. Also, it is recommended that you try to use at least two sections in your code, one for Chip RAM data and one for code. This will allow expanded systems and A4000's to execute much faster. If you divide it into many sections (but keep important modules together), your program will also cope better under highly fragmented memory conditions.

New offset sizes for displacements can be used for performance boosts with a little planning. While the 68000 supported only 8-bit displacements, 16-bit and 32-bit sizes are now available. Furthermore, the 16-bit displacement requires the same execution time if it is within the cache (it eats an extra clock cycle otherwise). The 32-bit displacements require 4 extra cycles. Also, it is generally faster to use address registers for addressing, since data registers are slower by three cycles than address registers, and it only takes two cycles to copy a data register to an address register. Nonetheless, data registers can be handy if there are no address registers free.

Conclusion

It is hoped that this information will serve as an encouraging guide for increased AA development. There is a strong need for a new breed of programs that take advantage of the new features and standards it offers. Don't be afraid to be something of a pioneer, there is plenty of marketspace for AA products, especially with the introduction of CD^{32}. Besides, the only way more people will jump the gap is if there is AA software that impresses them enough to attract them over. If that software is yours, then it is you who will gain the edge. The Amiga has matured once more, and it is time to start taking advantage of the many new benefits it offers, which this article cannot begin to describe. After all, the inclusion of a standard minimum 68020 processor is but one small facet of the changes and additions that have been made, and is more or less an incidental one. This truly is just the beginning.

As told by AC Tech #3.4 and Amiga World Aug. '93...

One Company Still Supports The Amiga!

One Amiga language has stood the test of time.

This new package represents the fourth major upgraded release of F-Basic since 1988. Packed with new features, 5.0 is the fastest and fullest yet. The power of C with the friendliness of BASIC. Compatibility with all Amiga platforms through the 4000...compiled assembly object code with incredible execution times... features from all modern languages, an AREXX port, PAL and ECS/AGA chip set support...Free technical support... This is the FAST one you've read so much about!

F-BASIC 5.0™

Supports DOS 1.3, 2.0, 2.1 and 3.0

F-BASIC 5.0™ System $99.95
Includes Compiler, Linker, Integrated Editor Environment, User's Manual, & Sample Programs Disk.

F-BASIC 5.0™ + SLDB System $159.95
As above with Complete Source Level DeBugger.

Available Only From: DELPHI NOETIC SYSTEMS, INC. **(605) 348-0791**
P.O. Box 7722 Rapid City, SD 57709-7722
Send Check or Money Order or Write For Info. Call With Credit Card or C.O.D.
Fax (605) 343-4728 Overseas Distributor Inquiries Welcome

Listing

```
Rem the AMIGA_PI program in AMOS ... 15 March 1994
Rem by Robert Davis at the Boarding House BBS (913) 827-0744
Rem the original idea for this program comes from
Rem "The Best of MICRO" Volume 1, pages 85-86, by Robert Bishop
Set Buffer 8
Screen Open 4,640,200,4,Hires
Palette 0,$177,$F0,$FFF : Rem R G B values (0 1 2 3)
Rem colour 1 = background, colour 2 = text
Erase 10 : Erase 11 : Erase 12 : Rem clear memory banks
Dim CONSTANT(2) : CONSTANT(1)=25 : CONSTANT(2)=239
TEN=10 : LINE=2 : CULUMN=4 : Curs Off
Locate CULUMN,LINE
Print "A PI calculation exercise in AMOS Basic." : Inc LINE : Inc LINE
Locate CULUMN,LINE
Print "Use right mouse button to pull down the menu." : Inc LINE
Rem the menu structure
Menu$(1)=" AMIGA_PI     "
```

```
Menu$(1,1)=" About ...          "
Menu$(1,2)=" Output to screen  "
Menu$(1,3)=" Output to printer "
Menu$(1,4)=" Quit              "
MENSTRT:
Wind Save : Rem remember what the window covers
Menu On
Do
  If Choice Then Exit
Loop
Menu Off
On Choice(2) Goto ABTPGM,SCRN,PRTR,PGMXIT
Stop : Rem error trap
ABTPGM:
Wind Open 5,16,32,72,10,1
Curs Off
Print " Original program for Apple II by Robert Bishop"
Print " Published in 'Best of Micro', volume 1, pp 85-86"
Print
Print " Amiga adaptation by Robert Davis, Salina, KS."
Print " rdavis@nyx.cs.du.edu  or  The Boarding House BBS 913 827-0744"
Print
Print " Press a key to continue."
Wait Key
Wind Close
Goto MENSTRT : Rem loop until user makes choice
PRTR: OWTPRTR=1 : Goto HWMANY
SCRN: OWTPRTR=0
HWMANY:
Inc LINE : Locate CULUMN,LINE : Curs On
Print "To how many digits shall we calculate PI";
Input SIZE : Curs Off
Inc LINE : Locate CULUMN,LINE
Print "Working, please wait."
On Error Goto GENERICERR
MEMREQUIRED=4*SIZE
Reserve As Data 10,MEMREQUIRED : Rem save some space in RAM
Reserve As Data 11,MEMREQUIRED
Reserve As Data 12,MEMREQUIRED
Rem error checks should go here
POWER=Start(10) : TERM=Start(11) : RESULT=Start(12) : Rem the memory banks
names
Rem program main loop
 For PASS=1 To 2
    Gosub INIT
INLP: Rem interior loop
    Gosub COPEE
    POYNT=TERM : DIVIDE=XPN : Gosub DIV
    If SIGN>0 Then Gosub ADX
    If SIGN<0 Then Gosub SBX
    XPN=XPN+2 : SIGN=-SIGN
    POYNT=POWER : DIVIDE=CONSTANT(PASS) : Gosub DIV
    If PASS=2 Then Gosub DIV
    If ZERO<>0 Then Goto INLP
 Next PASS
Rem output the result
If OWTPRTR=1 Then Goto DUPRTR
Inc LINE
Locate CULUMN,LINE
Print "The value of PI to";(TEN/100+1)*SIZE;" decimal places ..." : Print
Print Peek(RESULT);".";
For PLACE=RESULT+1 To RESULT+SIZE
  Print Right$(Str$(Peek(PLACE)),1);
Next PLACE
Print : Print : Print "    The last one or two digits are always suspect."
Goto PGMFIN
DUPRTR: Rem do printer
PRCHR=2 : Rem we print '3.' before starting the count
Lprint " The value of PI to";(TEN/100+1)*SIZE;" decimal places." : Lprint
Lprint Right$(Str$(Peek(RESULT)),1);".";
For PLACE=RESULT+1 To RESULT+SIZE
  Lprint Right$(Str$(Peek(PLACE)),1); : Inc PRCHR
  If PRCHR=76
     Lprint
     PRCHR=0
  End If
Next PLACE
Lprint : Lprint " The last one or two digits are always suspect."
Lprint Chr$(12);
PGMFIN: Rem the program is finished now
Print : Print "    Press a key to exit."
Wait Key
Stop : Rem internal end of program
Rem the DIV divison routine
DIV:
DIGIT=0 : ZERO=0
For PLACE=POYNT To POYNT+SIZE
  DIGIT=DIGIT+Peek(PLACE)
  QUOTIENT=DIGIT/DIVIDE
  RESIDUE=DIGIT mod DIVIDE
  ZERO=ZERO or(QUOTIENT+RESIDUE)
  Poke PLACE,QUOTIENT
  DIGIT=TEN*RESIDUE
Next PLACE
Return
Rem the ADX addition subroutine
ADX:
CARRY=0
For PLACE=SIZE To 0 Step -1
  SUM=Peek(RESULT+PLACE)+Peek(TERM+PLACE)+CARRY
  CARRY=0
  If SUM<TEN Then Goto OWTADX
  SUM=SUM-TEN
  CARRY=1
OWTADX:
  Poke RESULT+PLACE,SUM
Next PLACE
Return
Rem the SBX subtraction routine
SBX:
LOAN=0
For PLACE=SIZE To 0 Step -1
  DIFFERENCE=Peek(RESULT+PLACE)-Peek(TERM+PLACE)-LOAN
  LOAN=0
  If DIFFERENCE=>0 Then Goto OWTSBX
  DIFFERENCE=DIFFERENCE+TEN
  LOAN=1
OWTSBX:
  Poke RESULT+PLACE,DIFFERENCE
Next PLACE
Return
INIT: Rem initialize memory banks and variables
  For PLACE=0 To SIZE
    Poke POWER+PLACE,0
    Poke TERM+PLACE,0
    If PASS=1
       Poke RESULT+PLACE,0
    End If
  Next PLACE
    Poke POWER,16/(PASS*PASS)
    If PASS=1
       DIVIDE=5
    Else
       DIVIDE=239
    End If
    POYNT=POWER : Gosub DIV
    XPN=1 : SIGN=3-2*PASS
Return
Rem the COPY routine .. copy 'power' into 'term'
COPEE:
For PLACE=0 To SIZE
Poke TERM+PLACE,Peek(POWER+PLACE)
Next PLACE
Return
GENERICERR:
Print : Print " Generic error message."
Print " Press a key to exit."
Wait Key
PGMXIT:
End
```

✔

Please Write to:
Christopher Jennings
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

True BASIC AmigaDOS Module

by T. Darrel Westbrook

The lack of AmigaDOS functions to make a directory, read a directory, etc. is a major limitation to wide spread use of Amiga True BASIC. The Amiga True BASIC Student Edition, V2.0, has a library file named highdos*. Unfortunately, there is no Student Edition documentation to help you use the library in your programs. True BASIC, Inc. would not grant permission to reprint their highdos* source code, so I wrote my own version of AmigaDOS functions. This article will not only provide you with a powerful modular library, but it will show you, in detail, how to use True BASIC to access system level functions.

The AmigaDOS module is for use on OS 2.0 and higher. Several function calls are OS 2.0 specific. Line numbers are for reference only. There is an AmigaDOS Module demonstration program on the accompanying disk. To run the demonstration, run the AmigaDOS.demo program included on the enclosed disk. Now to continue with the analysis of the AmigaDOS module.

The True BASIC AmigaDOS module consists of eight functions and four subroutines (See Listing 1). They are;

Functions	**Subroutines**
ChDir	DirSort
GetDir$	Ch_DOS_Flag
GetRoot$	Dir_List
Get_PathPart$	Get_DOS_Param
MakeDir	
PathExist	
RenFile	
VolumeRelabel	

I use several functions, RenFile() and VolumeRelabel(), and one subroutine, DirList$(), to discuss programming methods that permit True BASIC access to the Amiga system routines. During our discussion you will notice I use the DOS library function Relabel() in the AmigaDOS Module VolumeRelabel() function. The module function had to be a different name than the system library routine. VolumeRelabel seemed appropriate.

A few definitions are in order before we get to the discussion of how the AmigaDOS Module works. I'll offer my definitions of structures, functions, and pointers.

A structure is to the C programming language like record file is to True BASIC. When I initialize a record file in True BASIC, I must supply the record size. Once I establish the record size, I cannot change it until the file is empty. What this does is set aside a record size amount of disk space (or memory) for each record in the file. The C structure works the same way, one record (i.e., structure) at a time. There is more to C programming structures than what I have presented, but it is not essential to understanding how True BASIC can construct structures to pass to the operating system.

A function processes arguments passed to it by the calling program. Once the function completes its processing, it returns a value to the calling program. The value it returns depends on the type of function you called. The functions used in the AmigaDOS Module are boolean (BOOL type), void (VOID type), and pointer type.

A boolean function returns a TRUE or FALSE result. TRUE is generally one and FALSE is zero. An example of a boolean function is the Examine() function. It returns a TRUE or FALSE regarding the success of its operation. I used the variable success throughout the module to show that the function results are boolean.

The VOID type doesn't return anything of value. Throughout the AmigaDOS Module, I assigned the variable void to function values when its value was unimportant or it does not return any values. An example of a VOID function is the FreeDosObject() function.

The last type of result returned by a function is a pointer. A good example of a function that returns a pointer is the AllocDosObject() function. If it is successful it returns a memory address greater than zero. If for some reason the function could not return a memory address (i.e., no more memory is available) then the function will return a zero. NULL is zero in the C programming language. Understanding pointers is important to understanding how True BASIC uses information returned by system functions.

A pointer is a memory address where something is stored. Think of this as a box that is eight bits (which is a BYTE), 16 bits (which is a WORD), or 32 bits (which is LONG) in size. The bits are numbered from right to left. Amiga addresses are 32 bits. The following 32 bit string has the fifth and twentieth bit set to one.

```
00000000 00001000 00000000 00010000
```

In the C language, counting starts at zero, so the fourth and nineteenth bit would be set. When you use C programming headers (designated with the file extension .h) to extract constants, exercise caution when the headers specify shifted bit values (i.e., 1 << 10).

The pointer could be anything. It could be a binary representation of a number, the start of a string, the start of a structure or it could be another pointer. You, the programmer, must know what type of information that is pointed to. If you want more information on pointers, an article in Amazing Computing V4.1, 1989, by Forest W. Arnold is a good place to start.

True BASIC's access to system functions is through a system call made by a subroutine in the amiga* library. This library is on disk one of the Amiga True BASIC Student Edition. On disk two of the Student Edition is a text file, SYSTEM_LIBRARIES.DOC. This text file provides a lot of information about the various functions found in the amiga* library. The amiga* library is the glue that ties True BASIC to the operating system. The next obvious question is what does it glue together.

In the Student Edition, disk one, AmigaTools drawer you will find compiled system library files. I generated these compiled files using a True BASIC program named genlib. Unfortunately, the genlib program is not on the Amiga Student Edition disk. The genlib that I have is from the original Amiga True BASIC release. If you want the genlib program, you will have to write to True BASIC, Inc. for the program.

The distribution disk contains a compressed file with the library files that I created from the function definition files of the Dillion Integrated C Environment (DICE) programming language. I added an underscore to the libraries that I generated and compiled so I could tell the difference from the libraries issued with the Student Edition (current as of OS 2.04) and the libraries I generated from the DICE FD files (more current than OS 2.04). It would be helpful to appreciate how the genlib works and what is in the function definition files.

For True BASIC to use an Amiga system function call, it has to find the function in the system. This is accomplished by getting the library base address of the library that contains the function. From the library base, I then jump to an offset from the library base. This last address holds the final address of the actual function. All this jumping around might seem confusing. Maybe another approach might clarify all this addressing.

Suppose I wanted to find you, who represent a function that can do some work. You live somewhere in the United States. Imagine the library base address equates to the state in which you live out of all the states in the United States. The offset from the library base address equates to the city in which you live, and the address at the offset location is the address of your house. Once I find your house, then I can locate you, if you're at home. The library base address can change every time it is loaded for system use. But the addresses (library base address to library offset to function) remain constant, relative to each other. This allows us to change the function code (like adding a room to your house) without changing its street address. You are now armed with the information you will need to grasp how all this ties together and allows True BASIC to use Amiga system functions.

In a very tight nutshell I have shown you how to use Amiga system routines in your True BASIC programs.

We'll start with the function definition file and how the True BASIC genlib program generates a library of functions. A function definition (FD) file looks like the following;

Sample Listing 1

```
* "dos.library"
##base _DOSBase
   .
   .
##bias 492
##public
   .
   .
Relabel(drive,newname)(d1/d2)
   .
   .
```

This is only a portion of the FD file. Check the disk for the compressed copy of the dos_def file. The ##bias xxx marker in the FD file sets or resets the library offset pointer. The next function immediately following the ##bias xxx gets the library offset address of -492. The offset is negative because the library base address is at the bottom of the function address (or more correctly, the library jump table). Figure 2 is the Amiga library structure example from the AMIGA ROM Kernel Reference Manual, Libraries.

Sample Listing 2

```
     Low Memory
 _________|__________
|   Jump Function N  |
|          .         |
|          .         |
|   Jump Function 3  |  - Jump Table values are
|  Jump Function 2  |        negative from LB
|  Jump Function 1  |
|___________________|  ____ Library Base (LB)
| Library Structure |
|___________________|
|     Data Area     |
|___________________|
          |
     High Memory
```

Each line in the FD file increases the jump table offset by four (4) bytes. The four bytes is the Amiga 32 bit address size. The AmigaDOS Relabel() function definition is -720 bytes from the library base address. The (d1/d2) represent registers, which are associated with the x1 and x2 in the True BASIC Relabel() function (See Sample Listing 3).

Sample Listing 3

```
DEF Relabel(x1,x2)
    CALL Packb(s$,1,32*3,0) ! make string size 96 bits
    CALL Packb(s$,1,32,4)   ! load library base address
    CALL Packb(s$,33,32,x1) ! load pointer, current name
    CALL Packb(s$,65,32,x2) ! load pointer, new name
    CALL s_call(-720,6,s$)  ! offset -720 bytes
    LET Relabel = UnPackB(s$,1,-32)
END DEF
```

same by adding chr$(0) to the end of the string in question. Almost all strings passed to the Amiga system functions must be NULL terminated. Addr() is another True BASIC function that returns the address of a string. You will pass these addresses to the function (remember they only understand address), which is done in the last line of our sample code above. Once your True BASIC program calls the function, its parameters are passed to the function of the Sample Listing 3 above.

The first line of the function, CALL Packb(s$,1,32*3,0), allocates a block of memory 96 bits in length filled with zeros and called s$. This 96 bits will hold three 4-byte addresses. The first address is the library base address, the second is oldvolumename, and the third is newvolumename. Whoa! Where in the world did library base address come from? The library base addresses for Exec, DOS, graphics, and intuition are in the amiga* library. If you use any other library besides these four, you must include the base address as the last argument in the function call. The AmigaDOS Module does not require the use of other libraries, so I won't discuss this particular point further. Suffice to say that you can get the address of any library operating within the Amiga library structure.

I have supplied you with AmigaDOS tools that allow access to AmigaDOS and should add new capabilities to your True BASIC programs and ease some of your programming code time.

The C language code for using the Relabel() function is;

```
success = Relabel(oldvolumename, newvolumename)
```

where success is TRUE or FALSE (i.e., one or zero), oldvolumename and newvolumename are self-explanatory. When you want to use the True BASIC Relabel() function you call it like any other True BASIC function.

```
LET success = Relabel(oldvolumename, newvolumename)
```

The only difference is the True BASIC keyword LET.

Now we'll examine the insides of the True BASIC function and see how it works. Remember, functions want addresses that point to the information they need. So, the information we pass to functions is the location in memory of the information.

For my example consider oldvolumename is TrueBASIC and newvolumename is MyTrueBASIC. The following True BASIC code would set up the information to pass to the Relabel() function.

```
LET oldvolumename$ = Nullt$("TrueBASIC:")
LET oldvolumename  = Addr(oldvolumename$)
LET newvolumename$ = Nullt$("MyTrueBASIC")
LET newvolumename  = Addr(newvolumename$)
LET success = Relabel(oldvolumename, newvolumename)
```

The Nullt$() True BASIC function adds a NULL character (i.e., chr$(0)) to the end of the old and new names. You could do the

In the second code line of the Sample Listing 3 above, the 4 is understood by genlib as the location of the DOS library base address. If this number is 8, it loads the graphics library address, 12 the Exec library base, 16 for Intuition library base, and zero for any other library base. The next two code lines of Sample Listing 3 load the old and new names addresses. This information is given to the s_call() subroutine (part of the amiga* library), which has the library offset. Once the function has completed its job, the first byte of s$ will contain the results of the function. It is important to note that you cannot use any system function which returns more than one byte of information. For example, suppose a function returned the 4-byte address of a 10KB data record (not likely, but for argument's sake), you would be able to access all the information in the 10KB record by using the 4-byte address. However, if the function returned two 4-byte addresses, both required to access the 10KB record, you will not be able to use this particular function.

As mentioned earlier, I have included on the disk a compressed file of the newly generated library source code and the compiled versions of the libraries. These compiled files give you access to Amiga system functions. You will need an Amiga systems book to determine the format of a given system function. By format I mean the type of function and the values, if any, that it returns. I use the AMIGA ROM Kernel Reference Manual set.

The AmigaDOS Module uses only DOS library functions. Listing 1 is a library of the specific DOS functions used in the compiled dos_Special* library (See Listing 2). You could include the dos_Special* functions in the AmigaDOS Module. I kept them separate, so they would not clash with any other libraries if I inadvertently used the complete DOS library.

I believe we have examined the mechanics behind the Relabel() function in enough detail for you to appreciate how the AmigaDOS Module VolumeRelabel() function works. Now I'll discuss the RenFile() function and then the DirList$() subroutine.

If you examine the RenFile() function, you will notice that there isn't much to it. Lines 293 to 304 are the most probable error handling and lines 305 to 309 erase the icon file created by True BASIC when you create a new file. Taken in this light, you can see that it is easy to create True BASIC functions/subroutines that access Amiga system functions. You can also accomplish some complicated activities using these functions. The DirList$() subroutines is such an example.

The DirList$() subroutine (lines 95 to 161) requires the use of seven Amiga system functions, AllocDosObject(), Lock(), UnLock(), DupLock(), Examine(), CurrentDir(), ExNext(), and FreeDosObject(). It also uses a FileInfoBlock (see Listing 3). We are concentrating on the file or directory name (bytes 8 through 108), file size (bytes 124 through 127), and the type (file or directory, bytes 4 through 7). This information is initially stored in three strings, list$, size$, and this_type$. I delimited the data with a back slash symbol (lines 112, 114, and 122).

After the program reads the directory, it returns the data structures to the system and transfers the string information to arrays appropriately name filename$(,) and dir$(). Filename$(,) is sorted, when the sort flag is set, and contains the filename and the size of each file in the directory. The dir$() array contains the name of each sub-directory in the directory. If you wanted other information from the FileInfoBlock, just change the DirList$() subroutine to pull this information out of the data block. Let's assume you needed the fib_Comment. You would add the following code after line 123.

```
LET your_data$ = str$(UnPackB(fib$,144*8+1,-32))
```

The 144 comes from the size column of Listing 3. I multiplied by 8 because UnPackB() works on bits and not bytes. The plus one allows for the counting that starts at one for True BASIC and zero for the C language reference material.

In a very tight nutshell I have shown you how to use Amiga system routines in your True BASIC programs. I have supplied you with AmigaDOS tools that allow access to AmigaDOS and should add new capabilities to your True BASIC programs and ease some of your programming time.

Listing 1

```
   !
   ! *********************************************
   ! *       AmigaDOS Module, Version 1.0        *
   ! *  Copyrighted, 1993 by T. Darrel Westbrook *
   ! *             All Rights Reserved           *
   ! *********************************************
   !
 1 MODULE AmigaDos
   ! *********************************************
   ! * dos_special_* library generated with True  *
   ! * BASIC's genlib program and the file        *
   ! * definition files (.fd) that came with my   *
   ! * register Dillion Integrated C Environment, *
   ! * DICE.  Source for dos_special_* is on the  *
   ! * distributed disk.                          *
   ! * ========================================= *
   ! * The amiga* library is owned by True BASIC, *
   ! * Inc. and is on the True BASIC, Amiga       *
   ! * Student Edition, V2.0.  Ensure the path to *
   ! * amiga* is correct before you try to run    *
   ! * demo program.
   ! *********************************************
   !
 2 LIBRARY "dos_special_*"
 3 LIBRARY "amiga*"

 4 PUBLIC savepath$  ! save startup pathname for later use

 5 SHARE root$  ! root directory of current device
 6 SHARE starting_path$  ! full path to current directory
 7 SHARE sort_flag  ! if zero, no sort preformed
 8 SHARE info_flag  ! determines if '.info' files are
   included in DirList()

 9 SHARE LOCK_SHARED  ! read only access allowed
10 SHARE LOCK_EXCLUSIVE  ! no other access allowed
11 SHARE ACCESS_READ  ! Same as LOCK_SHARED
12 SHARE ACCESS_WRITE  ! same as LOCK_EXCLUSIVE
13 SHARE DOS_FIB  ! used by AllocDosObject()

14 PRIVATE DirSort
   !
   ! *************************************
   ! * Declare Functions for Module's Use *
   ! *************************************
   !
15 DECLARE DEF AllocDosObject  ! * creates a DOS object
16 DECLARE DEF Addr  ! returns the 32 bit address of a
   string
17 DECLARE DEF ChDir  ! changes directory
18 DECLARE DEF CreateDir  ! * Creates a new directory
19 DECLARE DEF CurrentDir  ! * create directory lock on
   present directory
20 DECLARE DEF DupLock  ! * Duplicates a lock
21 DECLARE DEF Examine  ! * Examine a file or diectory
   assiciated with a lock
22 DECLARE DEF ExNext  ! * Examine the next entry in a
   directory
23 DECLARE DEF FreeDosObject  ! Frees an object allocated
   by AllocDosObject
24 DECLARE DEF GetDir$  ! gets the present directory
25 DECLARE DEF GetRoot$  ! get the curent root directory
26 DECLARE DEF Get_PathPart$  ! strips off the file/
   directory at the end of the pathname, see function for
   limitations
27 DECLARE DEF IoErr  ! * return error info from the system
28 DECLARE DEF Lock  ! * locks a directory or file
29 DECLARE DEF MakeDir  ! Make new directory
30 DECLARE DEF Nullt$  ! NULL terminates a string
31 DECLARE DEF ParentDir  ! * Get the parent of a file lock
32 DECLARE DEF PathExist  ! determines if a dirrectory or
   file exist
33 DECLARE DEF PathPart  ! * returns the file directory
   name
```

```
34 DECLARE DEF Relabel  ! * change the volumn name of a
   volume
35 DECLARE DEF Rename  ! * renames a file or directory
36 DECLARE DEF RenFile  ! module function to rename file/
   dir
37 DECLARE DEF UnLock  ! * unlock a previously locked
   directory or file
38 DECLARE DEF UnNullt$  ! remove the NULL terminate from
   a string
39 DECLARE DEF VolumeRelabel  ! relabels a volume
   !
   ! ******************************
   ! *  LOAD MODULE SPECIFIC DATA  *
   ! ******************************
   !
40 LET LOCK_SHARED    = -2  ! Was SHARED_LOCK, but SHARED
   is a reserved word, read only access allowed
41 LET LOCK_EXCLUSIVE = -1  ! Was EXCLUSIVE_LOCK, no other
   access allowed
42 LET ACCESS_READ    = -2  ! same as LOCK_SHARED
43 LET ACCESS_WRITE   = -1  ! same as LOCK_EXCLUSIVE
44 LET DOS_FIB        =  2  ! to AllocDosObject to setup
   FileInfoBlock

45 LET savepath$ = GetDir$  ! get current directory path
46 LET root$     = savepath$[1:pos(savepath$,":")]
47 LET starting_path$ = savepath$  ! until it is changed
48 LET sort_flag  = 1  ! sort directory array, zero is no
   sort
49 LET info_flag  = 1  ! for DirList(), get '.info' files
   !
   ! ******************************
   ! * MODULE PRIVIATE SUBROUTINES *
   ! ******************************
   !
50 SUB DirSort(filename$(,),dir$())
       ! ******************************
       ! * SHELL SORT THE DIRECTORIES *
       ! ******************************
       !
51     LET N = size(dir$)  ! # of array rows
52     LET h = 1
53     DO  ! find an h that is bigger than N
54        LET h = 3 * h + 1
55     LOOP until h > N
56     DO
57        LET h = int(h / 3)  ! this is an integer
58        FOR i=h+1 to N
59            LET hold$ = dir$(i)
60            LET j = i
              ! hold i index until we find where it belongs
61            DO while ucase$(dir$(j-h)) > ucase$(hold$)
62               LET dir$(j) = dir$(j-h)
63               LET j = j - h
64               IF j =< h then EXIT DO
65            LOOP
66            LET dir$(j) = hold$
67        NEXT i
68     LOOP until h = 1
       ! ***********************
       ! * SHELL SORT THE FILES *
       ! ***********************
       !
69     LET N = size(filename$,1)  ! # of array rows
70     LET h = 1
71     DO  ! find an h that is bigger than N
72        LET h = 3 * h + 1
73     LOOP until h > N
74     DO
75        LET h = int(h / 3)  ! this is an integer
76        FOR i=h+1 to N
77            LET hold1$ = filename$(i,1)
78            LET hold2$ = filename$(i,2)
79            LET j = i
              ! hold i until it's place is found
80            DO while ucase$(filename$(j-h,1)) >
ucase$(hold1$)
81               LET filename$(j,1) = filename$(j-h,1)
82               LET filename$(j,2) = filename$(j-h,2)
83               LET j = j - h
84               IF j =< h then EXIT DO
85            LOOP
86            LET filename$(j,1) = hold1$
87            LET filename$(j,2) = hold2$
88        NEXT i
89     LOOP until h = 1
90 END SUB  ! end of 'DirSort'
   !
   ! ********************
   ! * MODULE SUBROUTINES *
   ! ********************
   !
91 SUB Ch_DOS_Flag(sort,info)
       ! ***********************************************
       ! * Change module flag values.                  *
       ! * ========================================== *
       ! * sort_flag = 1, then sort, else no sort      *
       ! * info_flag = 1, get '.info', else no '.info'  *
       ! ***********************************************
       !
92     IF sort = 0 or sort = 1 then LET sort_flag = sort
93     IF info = 0 or info = 1 then LET info_flag = info
94 END SUB  ! end of 'Ch_Sort_Flag'


95 SUB DirList(filename$(,),dir$())
       ! ***********************************************
       ! *  This subroutine makes a system lock on the  *
       ! *  current directory, then builds an array$ of *
       ! *  the filenames, variable filename$(,), and   *
       ! *  subdirectories, variable dir$().  The       *
       ! *  filename$ array has filename$(n,1) is the   *
       ! *  filename and filename$(n,2) is the size of  *
       ! *  the file in bytes.  The arrays are returned *
       ! *  unsorted.  If you want to sort them use the *
       ! *  Sort Module functions.                      *
                      *
       ! * ========================================== *
       ! *  If no files, then size(filename$,1) = 0     *
       ! *  If no directories, then size(dir$) = 0      *
       ! ***********************************************
       !
96     LET dosptr = AllocDosObject(DOS_FIB,0)  ! allocate
   a FileInfoBlock structure
97     LET list$, size$, type$ = ""
98     LET myLock = CurrentDir(0)  ! any value passed will
   return the lock value for the current directory
99     LET void     = CurrentDir(myLock)  ! put a lock on
   current dir
100    LET myLock   = DupLock(myLock)  ! duplicate the
   lock so I can use it
101    CALL Packb(dosptr$,1,32,dosptr)  ! pack fib pointer
   into string dosptr$
102    CALL ZeroString(fib$,260)  ! initialize the
   FileInfoBlock transfer space
103    LET success = Examine(myLock,dosptr)
104    IF success  <> 0 then  ! Examine got fib$ data
          ! ****************************************
          ! * list$      - file & directory names     *
          ! * this_size$ - current file/directory byte *
          ! *              size                        *
          ! * size$      - file size, directories are  *
          ! *              zero                        *
          ! * this_type  - < 0, then a file, > 0 then  *
          ! *                                    a dir *
          ! * ===================================== *
          ! * info_flag = 1, then load .INFO ext, else *
          ! *                do not load .INFO files   *
          ! ****************************************
105       DO
106          LET success = ExNext(myLock,dosptr)
107          IF success  = 0 then EXIT DO
108          CALL ToString(1,dosptr$,fib$)  ! get fib new
   data
109          LET dir_file$ = trim$(UnNullt$(fib$[8:116]))
110          IF info_flag = 0 AND
   ucase$(dir_file$[len(dir_file$)-4:len(dir_file$)]) =
   ".INFO" then
                ! NO Op for info file extentions
111          ELSE  ! load everything
112             LET list$ = list$ & "\" & dir_file$
113             LET this_size$ = str$(UnPackB(fib$,124*8+1,
   -32)/256)  ! unpack file size
114             LET size$ = size$ & "\" & this_size$  !
   continue to build the size string
115             LET this_type = UnPackB(fib$,4*8+1,-32)  !
   unpack 'fib_DirEntryType' variable
116             IF this_type < 0 then  ! name is a file
117                LET this_type$ = "file"
118             ELSE  ! name is a directory
119                LET this_type$ = "dir"
```

```
120              LET dcount = dcount + 1  ! record
   number of directories found
121           END IF
122           LET type$ = type$ & "\" & this_type$
123           LET count = count + 1  ! count number of
   entries
124        END IF
125     LOOP
126   END IF
127   LET void = FreeDosObject(myLock,dosptr)  ! release
   memory of DOS object
128   LET void = UnLock(myLock)  ! release the directory
   read lock
129   LET list$ = list$[2:len(list$)]  ! trim off
   beginning '\'
130   LET size$ = size$[2:len(size$)]  ! trim off
   beginning '\'
131   LET type$ = type$[2:len(type$)]  ! trim off
   beginning '\'
      ! ****************************************
      ! * LOAD ARRAY WITH DIRECTORY INFORMATION *
      ! ****************************************
      !
132   MAT dir$      = nul$(dcount)  ! resize filename$ to
   match data
133   MAT filename$ = nul$(count-dcount,2)  ! resize
   filename$ to match data
134   IF dcount = 0 AND count = 0 then EXIT SUB
      ! ******************************************
      ! * LOAD STRINGS INTO filename$(,) & dir$() *
      ! ******************************************
      !
135   LET count_dir, count_file = 0  ! initialize array
   counters
136   FOR n=1 to count-1  ! load all but the last item
137      LET end1 = pos(list$,"\")
138      LET end2 = pos(size$,"\")
139      LET end3 = pos(type$,"\")
140      IF type$[1:end3-1] = "dir" then  ! load in
   directory array
141         LET count_dir = count_dir + 1
142         LET dir$(count_dir) = list$[1:end1-1]
143      ELSE  ! load in the filename array
144         LET count_file = count_file + 1
145         LET filename$(count_file,1) = list$[1:end1-
   1]  ! load filename
146         LET filename$(count_file,2) = size$[1:end2-
   1]  ! load file size
147      END IF
148      LET list$ = list$[end1+1:len(list$)]  ! strip
   off used part of string
149      LET size$ = size$[end2+1:len(size$)]  ! strip
   off used part of string
150      LET type$ = type$[end3+1:len(type$)]  ! strip
   off used part of string
151   NEXT n
152   IF type$ = "dir" then
153      LET dir$(count_dir+1) = list$  ! load directory
   name
154   ELSE
155      LET filename$(count_file+1,1) = list$  ! load
   filename
156      LET filename$(count_file+1,2) = size$  ! load
   file size
157   END IF
158   IF sort_flag = 1 then  ! sort arrays
159      CALL DirSort(filename$,dir$)
160   END IF
161 END SUB  ! end of 'DirList'


162 SUB Get_DOS_Param(prog_root$,prog_starting_path$)
      ! ******************************
      ! * Returns module variables to  *
      ! * the calling program.  They   *
      ! * are the start up and root    *
      ! * directory.                   *
      ! ******************************
      !
163   LET prog_root$ = root$
164   LET prog_starting_path$ = starting_path$
165 END SUB  ! end of 'Get_DOS_Param'
    !
    ! *******************
    ! * MODULE FUNCTIONS *
    ! *******************
```

```
    !
166 DEF ChDir(filename$)
      ! ***********************************
      ! * Function makes filename$ the new  *
      ! * current directory.                *
      ! * =============================== *
      ! * Exceptions Used;                  *
      ! *    9008, No Such directory        *
      ! * =============================== *
      ! * ChDir = 0, FALSE, function failed *
      ! *       = 1, TRUE, function success *
      ! ***********************************
      !
167   LET success = PathExist(filename$)
168   CALL error_reset
169   IF success = 2 then  ! it is a directory
170      LET a$ = Nullt$(filename$)
171      LET dosptr   = AllocDosObject(DOS_FIB,0)  !
   allocate a FileInfoBlock structure
172      CALL Packb(dosptr$,1,32,dosptr)  ! pack fib
   pointer into string dosptr$
173      LET myLock  = Lock(Addr(a$),LOCK_SHARED)
174      LET success = Examine(myLock,dosptr)
175      LET ChDir = UnLock(CurrentDir(myLock))
176      LET void  = FreeDosObject(myLock,dosptr)
177      LET ChDir = 1  ! return TRUE
178   ELSE  ! it's not a directory
179      CALL make_error(9008,"")  ! no such directory
180      LET ChDir = 0  ! return FALSE
181   END IF
182 END DEF  ! end of 'ChDir'


183 DEF GetDir$
      ! ***********************************
      ! * Function returns the full pathname *
      ! *     of the current directory.     *
      ! ***********************************
      !
184   CALL error_reset
185   LET dosptr = AllocDosObject(DOS_FIB,0)  ! allocate
   a FileInfoBlock structure
186   LET pathname$ = ""
187   LET myLock = CurrentDir(0)  ! any value passed will
   return the lock value for the current directory
188   LET void     = CurrentDir(myLock)  ! now make it
   the current lock
189   LET myLock   = DupLock(myLock)  ! duplicate the
   lock so I can use it
190   CALL Packb(dosptr$,1,32,dosptr)  ! pack fib pointer
   into string dosptr$
191   CALL ZeroString(fib$,260)  ! initialize the
   FileInfoBlock transfer space
192   DO  ! until oot directory is found, i.e., success =
   0
193      LET success = Examine(myLock,dosptr)  ! get
   FileInfoBlock info
194      IF success = 0 then EXIT DO  ! zero when at the
   root directory
195      CALL ToString(1,dosptr$,fib$)  ! get fib new data
196      LET dir$ = trim$(UnNullt$(fib$[8:116]))  ! get
   the current directory
197      LET pathname$ = dir$ & "/" & pathname$  ! add it
   to the total pathname
198      LET newLock = ParentDir(myLock)  ! get the
   parnet of the current directory
199      IF newLock  = 0 then EXIT DO  ! zero when at the
   root directory
200      LET void = UnLock(myLock)  ! newLock not zero,
   so not at the root
201      LET myLock = newLock  ! not at root, do another
   loop
202   LOOP
      ! ****************************************
      ! * FULL PATHNAME TO THE CURRENT DIRECTORY *
      ! ****************************************
      !
203   LET void = FreeDosObject(myLock,dosptr)  ! release
   memory of DOS object
204   LET void = UnLock(myLock)  ! release the directory
   read lock
205   LET hold = len(dir$)+1
206   LET pathname$[hold:hold] = ":"  ! replace '/' with
   ':'
207   LET GetDir$ = pathname$
208 END DEF  ! end of 'GetDir$'
```

```
209 DEF Get_PathPart$(full_path$)
      ! ************************************************
      ! * Function returns the pathname.  Actually, it *
      ! * finds the last "/" in a pathname and returns *
      ! * the string to the left of the last "/".  So, *
      ! * if full_path$ is 'MyDrive:TrueBasic/Dir',    *
      ! * and 'Dir' is actually a directory, the       *
      ! * function will return 'MyDrive:TrueBasic'.    *
      ! * If 'MyDrive:TrueBasic/Dir/', is passed as    *
      ! * full_path$, then the function returns        *
      ! * 'MyDrive:TrueBasic/Dir'.                     *
      ! ************************************************
      !
210     LET path$ = Nullt$(trim$(full_path$))  ! trim off
   white space
211     LET path  = addr(path$)
212     LET Get_PathPart$ = path$[1:(PathPart(path)-path)]
213 END DEF  ! end of 'Get_PathPart$'


214 DEF GetRoot$
      ! ******************************
      ! * Function returns the current *
      ! * root directory             *
      ! ******************************
      !
215     LET hold$ = GetDir$
216     LET x = pos(hold$,":")
217     LET GetRoot$ = hold$[1:x]
218 END DEF  ! end of 'GetRoot'


219 DEF MakeDir(Dir_Name$)
      ! ******************************************
      ! * Function makes, in the current directory, *
      ! * a new directory named, Dir_Name$.        *
            *
      ! * ======================================== *
      ! * IoErr = 103, error 9006, "Disk Full."     *
      ! * IoErr = 202, error 9008, "No such         *
      ! *                          directory."      *
      ! * IoErr = 212, error 7104, "Wrong type of   *
      ! *                          file."           *
      ! * IoErr = 214, error 9001, "File is read or *
      ! *                          write
      ! *                          protected."      *
      ! * IoErr = 218, error 9010, "Device not      *
      ! *                          mounted."        *
      ! * IoErr = ???, error 9009, "Directory       *
      ! *                          already exist."  *
      ! * ======================================== *
      ! * MakeDir = 0, FALSE, function failed      *
      ! *         = 1, TRUE, function success      *
      ! ******************************************
      !
220     CALL error_reset
221     LET myLock = Createdir(Addr(Nullt$(Dir_Name$)))
222     IF myLock = 0 then  ! CreateDir didn't work
223        SELECT CASE IoErr
224        CASE 103
225             CALL make_error(9006,"")
226        CASE 202  ! directory in use
227             CALL make_error(9008,"")
228        CASE 212  ! wrong type of file
229             CALL make_error(7104,"")
230        CASE 213,214  ! file is read or write protected
231             CALL make_error(9001,"")
232        CASE 218  ! device not mounted
233             CALL make_error(9010,"Device is not
   mounted.")
234        CASE else
235             CALL make_error(9009,"Directory already
   exist.")
236        END SELECT
237        LET MakeDir = 0  ! return FALSE
238     ELSE  ! directory created
239        LET void = UnLock(myLock)
240        LET MakeDir = 1  ! return TRUE
241     END IF  ! end of 'IF myLock = 0 ...
242 END DEF  ! end of 'MakeDir'
```

```
243 DEF PathExist(FileName$)
      ! ****************************************
      ! * Function determines if FileName$ (which *
      ! * is a full pathname) is a file, a       *
      ! * directory, or it doesn't exists.       *
      ! * ==================================== *
      ! * IoErr = ???, error 9000, "Unknown file  *
      ! *                           error."      *
      ! * IoErr = 202, error 9010, "File is in    *
      ! *                          Use."         *
      ! * IoErr = 205, error 9003, no such file   *
      ! * IoErr = 212, error 7104, "Wrong type of *
      ! *                           file."       *
      ! * ==================================== *
      ! * PathExist = 0, Not a file or directory  *
      ! *           = 1, valid filename          *
      ! *           = 2, valid directory         *
      ! ****************************************
      !
244     CALL error_reset
245     LET PathExist = 0  ! initialize
246     LET myLock = Lock(Addr(Nullt$(FileName$)),
   LOCK_SHARED)
247     IF myLock  = 0 then
248        SELECT CASE IoErr
249        CASE 212  ! object wrong type
250             CALL make_error(7104,"")
251        CASE 202  ! file in use
252             CALL make_error(9010,"File is in use.")
253        CASE 205
254             CALL make_error(9003,"")  ! No such file.
255        CASE else
256             CALL make_error(9000,"Unknown file error.")
257        END SELECT
258        EXIT DEF
259     END IF
260     CALL ZeroString(fib$,260)
261     LET success = Examine(myLock,Addr(fib$))
262     IF success  = 0 then  ! Examine failed, check IoErr
   for cause
263        SELECT CASE IoErr
264        CASE 205
265             CALL make_error(9003,"")  ! No such file.
266        CASE else
267             CALL make_error(9000,"Unknown file error.")
268        END SELECT
269        LET void = UnLock(myLock)  ! unlock the file for
   use
270        CALL make_error(9002,"")
271        EXIT DEF
272     END IF
273     LET void = UnLock(myLock)
274     IF UnPackB(fib$,33,-32) < 0 then  ! it is a file
275        CALL make_error(1,"Valid filename.")  ! pathname
   is a valid filename
276        LET PathExist = 1
277     ELSE  ! it's a valid directory
278        CALL make_error(2,"Valid directory.")  !
   pathname is a directory
279        LET PathExist = 2
280     END IF
281 END DEF  ! end of 'PathExist'


282 DEF RenFile(oldfile$,newfile$)
      ! ******************************************
      ! * Must enter with the full pathname for the *
      ! * file you want to rename.  It will not    *
      ! * rename across devices.                   *
      ! * ====================================== *
      ! * IoErr = 205, error 9003, "Directory      *
      ! *                          already exist." *
      ! * IoErr = 215, error 9012, "Renaming across *
      ! *                          devices."       *
      ! * IoErr = ???, error 9000, "Unknown file    *
      ! *                          error."         *
      ! * IoErr = ???, error 9013, "No such file or *
      ! *                          directory."     *
      ! * ====================================== *
      ! * RenFile = 0, FALSE, function failed      *
      ! *         = 1, TRUE, function success      *
      ! ******************************************
      !
```

```
    LET oldfile$ = trim$(oldfile$)  ! trim off access white space
    LET success  = PathExist(oldfile$)
    IF success = 0 then  ! no file/directory
       CALL make_error(9013,"No such file or directory.")
       LET RenFile = 0  ! return FALSE, function failed
       EXIT DEF
    END IF
    LET a$ = Nullt$(oldfile$)
    LET b$ = Nullt$(newfile$)
    LET success = Rename(Addr(a$),Addr(b$))
    IF success = 0 then  ! some error occured
       SELECT CASE IoErr
       CASE 205  ! object not found
            CALL make_error(9003,"")  ! No such file.
       CASE 215
            CALL make_error(9012,"Renaming across devices.")
       CASE else
            CALL make_error(9000,"Unknown file error.")
       END SELECT
       LET RenFile = 0  ! return FALSE, some eror has occured
       EXIT DEF
    END IF
    ! ***********************************
    ! * Rename any .INFO files associated *
    ! * with the old filename            *
    ! ***********************************
    !
    IF PathExist(oldfile$ & ".info") <> 0 then  ! either file/dir .info file
       LET a$ = Nullt$(oldfile$ & ".info")
       LET b$ = Nullt$(newfile$ & ".info")
       LET success = Rename(Addr(a$),Addr(b$))
    END IF
    LET RenFile = 1  ! return TRUE, Rename was a success
END DEF  ! end of 'RenFile'


DEF VolumeRelabel(oldlabel$,newlabel$)
    ! ***********************************************
    ! * Function relabels a volume.  oldlabel$ must *
    ! * have the trailing ':' and newlabel$ must    *
    ! * not have the ':'.  Once function is         *
    ! * completed, it returns to the directory it   *
    ! * originally started from.  If the current    *
    ! * directory is a device that you are relabel- *
    ! * ing, then you will get a System requester.  *
    ! * Just cancel the requester and the program   *
    ! * will continue.                              *
    ! * =========================================== *
    ! * Exceptions Used;                            *
    ! *    9001, File is read/write protected       *
    ! *    9005, Diskette removed                   *
    ! *    9011, "Incorrect root directory format." *
    ! * =========================================== *
    ! * VolumeRelabel = 0, FALSE, function failed   *
    ! *               = 1, TRUE, function success   *
    ! ***********************************************
    !
    LET oldlabel$ = trim$(oldlabel$)
    LET newlabel$ = trim$(newlabel$)
    IF pos(oldlabel$,":") = 0 then LET oldlabel$ = oldlabel$ & ":"
    IF pos(oldlabel$,":") <> len(oldlabel$) then  ! invalid format
       CALL make_error(9011,"Incorrect root directory format.")
       LET VolumeRelabel = 0  ! return a FALSE, function failed
       EXIT DEF
    END IF
    ! **********************
    ! * Check for oldlabel$ *
    ! **********************
    !
    LET result = PathExist(oldlabel$)  ! ensure that oldlabel$ is valid
    SELECT CASE result
    CASE 2  ! it's a valid root directory
         LET a$ = Nullt$(oldlabel$)  ! NULL terminate the string
         LET b$ = Nullt$(newlabel$)  ! before using system function
         LET success = Relabel(Addr(a$),Addr(b$))
         IF success = 0 then  ! an error has occured
            SELECT CASE IoErr
            CASE 214,222,223,224  ! Disk write protected
                 CALL make_error(9001,"")  ! cause error, file is read or write protected
            CASE 226  ! no disk present
                 CALL make_error(9005,"")  ! cause error, no disk present
            CASE else
                 CALL make_error(9000,"Unknown file error.")  ! cause error, no disk present
            END SELECT
            LET VolumeRelabel = 0  ! return FALSE, function failed
         ELSE  ! success OK
            LET VolumeRelabel = 1  ! return TRUE, function success
         END IF  ! end of 'IF success <> 0 ...
    CASE else  ! it's a file or it doesn't exist
         LET VolumeRelabel = 0  ! return FALSE, function failed
    END SELECT
END DEF  ! end of 'VolumeRelabel'
END MODULE  ! end of AmigaDOS
```

Listing 2

```
!
!
! This True BASIC library file is a condensed version
! of the library generated by True BASIC's genlib program
! and Dillion's Integrated C Environment, DICE def files.
!

EXTERNAL

DEF Rename(x1,x2)
    CALL Packb(s$,1,32*3,0)
    CALL Packb(s$,1,32,4)
    CALL Packb(s$,33,32,x1)
    CALL Packb(s$,65,32,x2)
    CALL s_call(-78,6,s$)
    LET Rename = Unpackb(s$,1,-32)
END DEF

DEF Lock(x1,x2)
    CALL Packb(s$,1,32*3,0)
    CALL Packb(s$,1,32,4)
    CALL Packb(s$,33,32,x1)
    CALL Packb(s$,65,32,x2)
    CALL s_call(-84,6,s$)
    LET Lock = Unpackb(s$,1,-32)
END DEF

DEF UnLock(x1)
    CALL Packb(s$,1,32,4)
    CALL Packb(s$,33,32,x1)
    CALL s_call(-90,2,s$)
    LET UnLock = Unpackb(s$,1,-32)
END DEF

DEF DupLock(x1)
    CALL Packb(s$,1,32,4)
    CALL Packb(s$,33,32,x1)
    CALL s_call(-96,2,s$)
    LET DupLock = Unpackb(s$,1,-32)
END DEF

DEF Examine(x1,x2)
    CALL Packb(s$,1,32*3,0)
    CALL Packb(s$,1,32,4)
    CALL Packb(s$,33,32,x1)
    CALL Packb(s$,65,32,x2)
    CALL s_call(-102,6,s$)
    LET Examine = Unpackb(s$,1,-32)
END DEF

DEF ExNext(x1,x2)
    CALL Packb(s$,1,32*3,0)
    CALL Packb(s$,1,32,4)
    CALL Packb(s$,33,32,x1)
    CALL Packb(s$,65,32,x2)
    CALL s_call(-108,6,s$)
    LET ExNext = Unpackb(s$,1,-32)
END DEF

DEF CreateDir(x1)
    CALL Packb(s$,1,32,4)
    CALL Packb(s$,33,32,x1)
    CALL s_call(-120,2,s$)
    LET CreateDir = Unpackb(s$,1,-32)
END DEF

DEF CurrentDir(x1)
    CALL Packb(s$,1,32,4)
    CALL Packb(s$,33,32,x1)
    CALL s_call(-126,2,s$)
    LET CurrentDir = Unpackb(s$,1,-32)
END DEF

DEF IoErr
    CALL Packb(s$,1,32,4)
    CALL s_call(-132,0,s$)
    LET IoErr = Unpackb(s$,1,-32)
END DEF

DEF ParentDir(x1)
    CALL Packb(s$,1,32,4)
    CALL Packb(s$,33,32,x1)
    CALL s_call(-210,2,s$)
    LET ParentDir = Unpackb(s$,1,-32)
END DEF

DEF Execute(x1,x2,x3)
    CALL Packb(s$,1,32*4,0)
    CALL Packb(s$,1,32,4)
    CALL Packb(s$,33,32,x1)
    CALL Packb(s$,65,32,x2)
    CALL Packb(s$,97,32,x3)
    CALL s_call(-222,14,s$)
    LET Execute = Unpackb(s$,1,-32)
END DEF

DEF AllocDosObject(x1,x2)
    CALL Packb(s$,1,32*3,0)
    CALL Packb(s$,1,32,4)
    CALL Packb(s$,33,32,x1)
    CALL Packb(s$,65,32,x2)
    CALL s_call(-228,6,s$)
    LET AllocDosObject = Unpackb(s$,1,-32)
END DEF

DEF FreeDosObject(x1,x2)
    CALL Packb(s$,1,32*3,0)
    CALL Packb(s$,1,32,4)
    CALL Packb(s$,33,32,x1)
    CALL Packb(s$,65,32,x2)
    CALL s_call(-234,6,s$)
    LET FreeDosObject = Unpackb(s$,1,-32)
END DEF

DEF ExAll(x1,x2,x3,x4,x5)
    CALL Packb(s$,1,32*6,0)
    CALL Packb(s$,1,32,4)
    CALL Packb(s$,33,32,x1)
    CALL Packb(s$,65,32,x2)
    CALL Packb(s$,97,32,x3)
    CALL Packb(s$,129,32,x4)
    CALL Packb(s$,161,32,x5)
    CALL s_call(-432,62,s$)
    LET ExAll = Unpackb(s$,1,-32)
END DEF

DEF Relabel(x1,x2)
    CALL Packb(s$,1,32*3,0)
    CALL Packb(s$,1,32,4)
    CALL Packb(s$,33,32,x1)
    CALL Packb(s$,65,32,x2)
    CALL s_call(-720,6,s$)
    LET Relabel = Unpackb(s$,1,-32)
END DEF

DEF PathPart(x1)
    CALL Packb(s$,1,32,4)
    CALL Packb(s$,33,32,x1)
    CALL s_call(-876,2,s$)
    LET PathPart = Unpackb(s$,1,-32)
END DEF
```

Listing 3

```
**********************************************************
  Returned by Examine() and ExNext(), must be a 4 byte
  boundary

  struct FileInfoBlock {
     size -> 260
       0   LONG    fib_DiskKey;
       4   LONG    fib_DirEntryType;  /* Type of Directory.
If < 0, then a file
                                       If > 0 a directory
       8   char    fib_FileName[108]; /* Null terminated.
Max 30 chars used
                                       for now */
     116   LONG    fib_Protection;    /* bit mask of
                                       protection, rwxd
                                       are 3-0. */
     120   LONG    fib_EntryType;
     124   LONG    fib_Size;          /* Number of bytes in
                                       file */
     128   LONG    fib_NumBlocks;     /* Number of blocks
                                       in file */
     140   struct DateStamp fib_Date; /* Date file last
                                       changed */
             LONG   ds_Days;          /* Number of days
                                       since 1 Jan 78 */
             LONG   ds_Minute;        /* Number of minutes
                                       past midnight */
             LONG   ds_Tick;          /* Number of ticks
                                       past minute */
           }; /* end of DateStamp structure */
     144   char    fib_Comment[80];   /* Null terminated
                                       comment associated
                                       with file */
     224   char    fib_Reserved[36];
   };

Where LONG is 4 bytes and char denotes a character array.
 **********************************************************
```

C Structure of FileInfoBlock. Byte sizes added.

✔

Please Write to:
T. Darrel Westbrook
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

HUGE NUMBERS (Part 3)

by Michael Griebling

Last article, the initial CLI-based calculator was extended to support logical, bit, and shift operations; conversions to/from any numerical base were added; and an interface to the IEEE 64-bit math library was provided to give the calculator trigonometric, hyperbolic, logarithmic, and power functions.

This final article creates a user-friendly graphical interface (Figure 1) for the CLI calculator using the CanDo program. First, the calculator graphical interface is designed and constructed using CanDo; second, support software is defined which provides the calculator interface functionality; and finally, the graphical interface is integrated with the CLI calculator. I conclude with a demonstration of the new GUI-based (Graphical User Interface) calculator.

Building a Graphical Interface

Probably the most daunting activity when designing software has to be the creation of the graphical interface. User interfaces typically make up about 60 percent of the entire application's code and are very labor-intensive since they require many iterations of compile-link-execute cycles to get just the right placement of 'gadgets' (i.e., buttons and text fields). Fortunately, a program called CanDo (see back issues of Amazing/Amiga Computing) provides a simpler alternative to the otherwise painful process of defining a program's GUI.

I designed the basic arrangement of calculator keys (Figure 1) to give a central numeric cluster with less used keys positioned to the sides of the numeric cluster. This calculator arrangement is typical of most commercial calculators with the exception that I opted for a horizontally extended keypad instead of the more common vertically extended keypad to maximize the use of the available screen space and give more space to the equation display, located just above the keypad. To reduce typing (or clicking), a scrollable region is located just above the display area which shows all the previously entered equations. You click on one of these equations to bring it back into the equation display for editing or recalculation.

CanDo V2.01 provides a new tool called 'SuperDuper' which can duplicate a single gadget like a button or text field any number of times with both vertical and horizontal offsets. Thus, the above matrix of calculator keys is extremely simple to lay out. First, I defined a single button with dimensions of 50 by 15 pixels so that 54 of these buttons in a nine by six matrix would fill the window area. Using the SuperDuper tool with an x offset of 50 and y offset of 15, I duplicated this button to totally fill the window both vertically and horizontally. Buttons located in the central area were then deleted to make room for the central button cluster. Starting with a single button which was half the horizontal size of the function keys, I duplicated this button to create the six by six central numeric cluster.

The calculator key labels are composed of text strings written to the display using a custom font after the initial calculator window has been created. The Helvetica-like font contains special characters to allow display of radicals, exponents, and powers on the calculator keys. Unfortunately, CanDo doesn't support any font other than Topaz for use in a text field, so the equation field can't reproduce the key labels exactly. I was forced to substitute alphabetic abbreviations for the roots and powers. The equation recall list uses the same abbreviations as the equation field.

The result window/equation entry field consists of a left-justified text field outlined with a beveled border. Equations can be entered directly into this field via the keyboard or by a series of mouse clicks on the calculator keys.

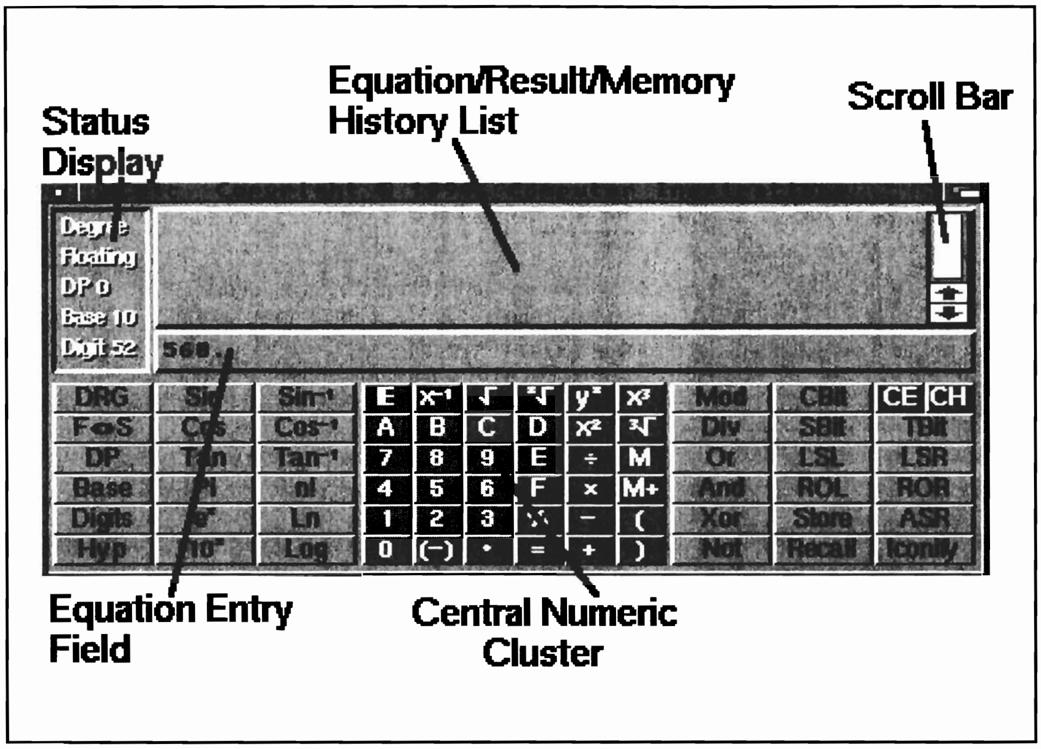


Figure 1
The calculator is designed with the basic arrangement of keys around a central numeric cluster with less used keys positioned to the sides.

Above the equation entry field is the equation/memory/result list which consists of a CanDo document tied to a scrollable list object. I update this list with either the most recently entered equation or a list of the current calculator contents or a history of the calculated results. The choice of what gets displayed is selectable via the History/Show menu items (more about the menus below). Only the displayed selection gets logged to the list object by calculations so when displaying the equations (Figure 2), for example, the previous calculated results (Figure 3) are not retained. Of course, memory locations (Figure 4) are kept current and will always display the exact calculator memory contents.

To the left of the history list is the calculator status display which shows the current angular measurement system, the type of floating point display (either floating decimal point or scientific notation), the number of decimal places, the numerical base (from 2 to 16), and the number of significant calculator digits in use. The five buttons immediately below this status display control these calculator attributes.

Three menu bar items: Project, Custom, and History give access to project items such as iconification, printing, and an 'about' box; item list display, clearing, and printing; and calculator customization options including selection of the processor-specific calculator (i.e., 68000, 68020, or 68030). The menu-based customization options duplicate the left-most calculator keys to a degree, although they are less flexible.

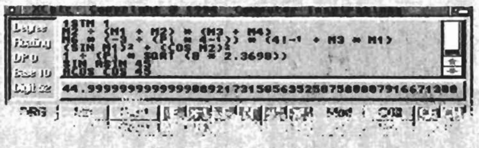
How The Calculator Interface Works

When I defined the calculator key labels (Listing 1) in the window startup script, I also added a string to the key which gets appended to the active equation string via a call to "ProcessKey" (Listing 2) when you click on a calculator key. This same string also gets displayed in the equation entry field. CanDo doesn't allow the setting of the cursor in the text field so, unfortunately, the cursor doesn't follow along when new text is appended via key clicks. When entering equations from the keyboard, everything works as you would expect.

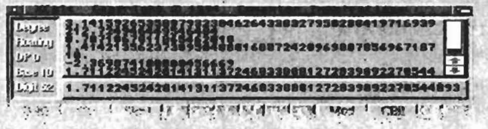
There are some complications to this approach. For instance, when you have entered a <Return> or clicked on the Equals key, you expect the following key clicks to start a new equation and not just append to the end of the equation which was just calculated. The "AddKeyToDisplay" routine encapsulates these intricacies as follows:

```
*************
* Global routine "AddKeyToDisplay"
        If First
                Let First = False
                SetText "Display",Arg1
        ElseIf (ArgCount > 1) And LastKeyNumber
                SetText "Display",Arg1||TextFrom("Display")
                Let LastKeyNumber = False
        Else
                SetText "Display",TextFrom("Display")||Arg1
        EndIf
        If FindChars("01234567890ABCDEF.",Arg1,1) = 0
                Let LastKeyNumber = False
        EndIf
        SetObjectState "Display",On
* End of routine "AddKeyToDisplay"
*************
```

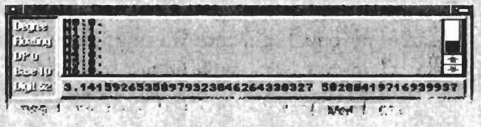
Here the "First" flag is set by other routines whenever the display should be cleared and a fresh equation is started (e.g., when the `=' key has been entered). The "LastKeyNumber" flag is used to prepend new strings to an existing equation in the case where a number was previously entered and then a function like `SIN' is selected via a button click. For example, if you just entered the number `45' and then clicked on the `SIN' key, the calculator equation would be `SIN 45' instead of `45 SIN'. This simple addition makes the calculator much more user-friendly since it automatically corrects a common mistake you might make.



Equation History Display



Numeric History Display



Memory Display

From Top to Bottom: Only the displayed selection gets logged to the list object by calculations so when displaying the equations (Figure 2), for example, the previous calculated results (Figure 3) are not retained. Of course, memory locations (Figure 4) are kept current and will always display the exact calculator memory contents.

Integrating the Calculator with CanDo

The heart of the calculator interface is the "Calculate" routine (Listing 3) which both submits the final equation string for computation to the external calculator and parses the entered equation in order to update the status display area.

As a first step, the equation is extracted from the "Display" text entry object and then passed to the "CallXCalc" routine to be processed as follows:

```
*************
* Global routine "CallXCalc"
            SetPointer Dir || "Brush/BusyCursor",5,5
            Dos State.Calculator||" >RAM:Result.txt " || Char(34) || Arg1
|| Char(34)
            OpenFile "RAM:Result.txt","ResultBuffer",READONLY ,OLDFILE
            FileReadLine "ResultBuffer",Arg2
            Close "ResultBuffer"
            SetPointer
* End of routine "CallXCalc"
*************
```

In this routine I am invoking the calculator defined in the "State.Calculator" field (i.e., either a 68000-, 68020-, or 68030-based calculator) and redirecting the calculator's normal output to a RAM-based text file called "Result.txt". I then open this file from within CanDo, read the result, and return the calculated result to the caller of this routine via the second argument `Arg2'. A busy pointer is also displayed as long as this routine is waiting for the external calculator to return an answer.

Note that CanDo's 'Dos' calling routine will wait until the calculator has calculated and placed the result in the output file and terminated its execution. I thus avoid synchronization problems in attempting to access the 'Result.txt' file which might otherwise occur if the CanDo script used the usual 'Dos Run' command sequence which does not wait.

To speed up the external calculator, I make the chosen calculator resident during CanDo's 'After Attachment' script with the following line:

```
    Dos "Run >Nil: <Nil: c:Resident " | | Dir | | State.Calculator
```

As a second step, the 'Calculate' routine updates the history display and parses the equation string to determine whether the status area needs to be updated with a new decimal point, numeric base, or the number of digits. This step is essential to guarantee that the CanDo status area always accurately reflects the state of the external calculator.

If an error has occurred (i.e., the key word "Illegal" is contained in the external calculator's result), no status or history update is allowed to prevent erroneous history and status values.

Printing from CanDo

One disappointment with CanDo is its lack of a built-in print command. Since I wanted to be able to print the history list contents, I defined the following "Print" routine:

```
*************
* Global routine "Print"
    OpenFile "RAM:Print.dat","PrintBuf",WRITEONLY
,NEWFILE
    FileWriteChars "PrintBuf",Arg1
    Close "PrintBuf"
    Dos "Copy RAM:Print.dat PRT:"; Print the above file & wait
* End of routine "Print"
*************
```

The argument to this routine ('Arg') is written to an external text file and then this file is copied to the printer port via a 'Dos' copy. I had to go this indirect route because I wanted the CanDo script to wait until the given file had been sent to the printer because the print routine is called multiple times when printing all the history information. I thus prevented the possibility that a subsequent print command would attempt to open the 'PRT:' device a second and even a third time while it was still busy. This bug actually occurred during development and I was scratching my head for quite a while until I finally discovered the root of this problem.

CanDo Menu Tricks

The CanDo menu system poses some additional challenges, especially when attempting to give programs the "look and feel" of the V2.0 operating system. I was forced to simulate the separating lines, typically seen dividing unlike menu items, by using a text string consisting of a series of hyphens which I then disabled so that the menu item takes on the familiar ghosted appearance and is inactive when you attempt to select it.

Another limitation is that mutually-exclusive menu items (in which a series of alternatives can each be checked but two or more are not allowed to be active simultaneously) are not explicitly supported by CanDo. To implement mutually-exclusive menu items, I defined a routine in CanDo like the following:

```
*************
* Global routine "SetDigitsMenu"
    SetObjectState "8",Off
    SetObjectState "12",Off
    SetObjectState "16",Off
    SetObjectState "20",Off
    SetObjectState "32",Off
    SetObjectState "40",Off
    SetObjectState "52",Off
    If Match(Arg1,"8","12","16","20","32","40","52") > 0
        SetObjectState Arg1,On
    EndIf
* End of routine "SetDigitsMenu"
*************
```

which set the state of all the listed menu buttons to an 'off' condition. For checked menu items this means that the checkmark is removed. The caller of this clearing routine then toggles an internal flag and sets its own button checkmark to either 'on' or 'off' depending on the internal flag. Using CanDo's built-in toggle buttons would have been easier but I couldn't always guarantee that the checkmark imagery would be synchronized with the actual toggle state of the button. Keeping an internal toggle state flag avoids any problems.

Installing the Calculator

To install the calculator on your system, decompress the contents of the accompanying disk and copy the 'XCalc' directory and all subdirectories to a disk or directory of your choice. It is important that the subdirectory structure and all the contained files are maintained in their respective locations when you copy this directory (i.e., use 'copy all'). If using the workbench, just drag the decompressed 'XCalc' drawer icon to the drawer where you wish to install the calculator. Double-click on the 'XCalc' calculator icon within the 'XCalc' drawer to invoke the calculator.

A Demonstration

Once the calculator is installed and displaying the interface shown in Figure 1, you can use the mouse to click on any calculator keys to enter the corresponding symbol in the display object. Alternatively, you can use the Amiga's keyboard to enter equations by clicking on the display object and then typing normally from the keyboard. You can switch back and forth between using mouse clicks and keyboard entry, keeping in mind that the calculator is case-sensitive and expects function names in uppercase characters. When in doubt, click on the desired function to get the proper capitalization for that operation.

If the equation history log has been enabled via the History/Display menu, all the equations which you enter are displayed in the history area. Other History/Display menu options show the calculation results or the current calculator memory contents. Select the equation option to keep track of the equation you'll enter. Click on the 'CE' button and enter the equation:

```
SQRT(3^2 + 4^2)
```

which computes the length of the hypotenuse of a right-angle triangle with sides of length 3 and 4. Click on the '=' or enter a <Return> and the answer '5.' should be displayed. Notice that this equation has also become the first history element in the history display. Click on the 'CE' button again and click on the equation in the history display. It should reappear in the equation display. Click on the display object around the '3^2' and use the keyboard cursor keys to position the cursor over the '3'. Replace the '3' with a '5' and similarly replace the '4' with a '12' to get the equation:

```
SQRT(4^2 + 12^2)
```

Click on the '=' key to get the new result of '13.'. Now click on the 'CH' and 'CE' buttons to clear the history display and clear the answer.

I will be using memory location 2 (M2) to store the number whose root is being found. Initialize this location with the number 100 as follows:

```
100 STM 2
```

Similarly initialize memory location 1 (M1) with an initial guess of the root, in this case the original number divided by 2:

```
M2 / 2 STM 1
```

Next enter the following equation which iteratively computes the square root of a number followed by an '=' (see the second article for details):

```
0.5 * (M1 + M2 / M1) STM 1
```

where M1 is the current estimate of the square root and M2 is the number whose root is being found. The first iteration should display the answer '26.'. Recall the equation from the history display and click on '=' again. The second iteration gives '14.923....'. Keep recalling the equation and clicking on '=' until the answer '10.'

is displayed after about four more iterations. From the History/ Display menu options select the display of the calculator memory contents. Memory locations 0 to 15 are displayed in the history display with M1 set to '10.' and M2 set to '100.'. Clicking on a memory location in the history window recalls the contents of that location.

For the last example, make sure that the answer '10.' is still displayed and click on the 'BAS' key. Now click on '1' then '6' to enter 'BAS 16'. Click on '='. The status display shows that the calculator is now operating with base 16 numbers and the previous result of '10.' is now displayed as the converted number '000A'. Click on 'CE' and enter the following equation:

(0'FFFF'0000'FFFF OR 1234'0000) AND 0'AAAA

from the keyboard and end with a <Return>. The answer '0FFFF1234AAAA' is displayed. Note the use of apostrophes to visually separate groups of four hexadecimal digits. This capability is extremely useful when entering very large numbers. The calculator also allows the use of a comma as a separator.

The memory contents have not been altered when the calculator switched bases. The reason for this is that fractional results could get truncated when working with numerical bases other than ten, so only numbers which are actually recalled from memory and used in a calculation are truncated.

Summary

I have just touched on a few of the calculator's capabilities in this brief demonstration. To fully appreciate all the calculator features, experiment on your own with the calculator and attempt some useful modifications of the supplied Oberon-2 source code or the CanDo decks which make up the user interface. Some suggestions are to give the CanDo interface the capability of storing in the history log both formulae and calculated results. The calculator could be extended with statistical functions and imaginary numbers. I would be interested in hearing about the uses you find for this calculator and any additions you make. Happy computing!

Listings

Listing 1

```
BeforeAttachment ; used to be OnStartup
   Nop; Various internal variables
   Let First = True
   Let Before = 0
   If Exists("RAM:CandoCalc.state")
        Let State = LoadVariable("RAM:CandoCalc.state")
   Else
        Let State.Display = ""
        Let State.DegRadGrad = "Degree"
        Let State.Notation = "Floating"
        Let State.DecimalPoint = 0
        Let State.Base = 10
        Let State.Digits = 52
        Let State.Calculator = "Calculator"
        Let State.Formulae = ""
        Let State.Numbers = ""
        Let State.Memory = ""
        Let State.UseBuffer = 2
EndIf
Nop; Calculator key definitions
Let Key[1,1].Label = "DRG"
Let Key[1,1].Text  = "DRG"
Let Key[1,2].Label = "F++S"
Let Key[1,2].Text  = "SCI"
Let Key[1,3].Label = "DP"
Let Key[1,3].Text  = "DP "
Let Key[1,4].Label = "Base"
Let Key[1,4].Text  = "BAS "
Let Key[1,5].Label = "Digits"
Let Key[1,5].Text  = "DIG "
Let Key[1,6].Label = "Hyp"
Let Key[1,6].Text  = ""
Do "DefineTrigKeys"
Let Key[2,4].Label = "Pi"
Let Key[2,4].Text  = "Pi"
Let Key[2,5].Label = "e*"
Let Key[2,5].Text  = "e^"
Let Key[2,6].Label = "10*"
Let Key[2,6].Text  = "10^"
Let Key[3,4].Label = "n!"
Let Key[3,4].Text  = "!"
Let Key[3,5].Label = "Ln"
Let Key[3,5].Text  = "LN "
Let Key[3,6].Label = "Log"
Let Key[3,6].Text  = "LOG "
Let Key[4,1].Label = "«"
Let Key[4,1].Text  = "E"
Let Key[4,2].Label = "A"
Let Key[4,2].Text  = "A"
Let Key[4,3].Label = "7"
Let Key[4,3].Text  = "7"
Let Key[4,4].Label = "4"
Let Key[4,4].Text  = "4"
Let Key[4,5].Label = "1"
Let Key[4,5].Text  = "1"
Let Key[4,6].Label = "0"
Let Key[4,6].Text  = "0"
Let Key[5,1].Label = "x+á"
Let Key[5,1].Text  = "+á"
Let Key[5,2].Label = "B"
Let Key[5,2].Text  = "B"
Let Key[5,3].Label = "8"
Let Key[5,3].Text  = "8"
Let Key[5,4].Label = "5"
Let Key[5,4].Text  = "5"
Let Key[5,5].Label = "2"
Let Key[5,5].Text  = "2"
Let Key[5,6].Label = "(-)"
Let Key[5,6].Text  = "-"
Let Key[6,1].Label = "é"
Let Key[6,1].Text  = "SQRT "
Let Key[6,2].Label = "C"
Let Key[6,2].Text  = "C"
Let Key[6,3].Label = "9"
Let Key[6,3].Text  = "9"
Let Key[6,4].Label = "6"
Let Key[6,4].Text  = "6"
Let Key[6,5].Label = "3"
Let Key[6,5].Text  = "3"
Let Key[6,6].Label = "?"
Let Key[6,6].Text  = "."
Let Key[7,1].Label = "*é"
Let Key[7,1].Text  = " ROOT "
Let Key[7,2].Label = "D"
Let Key[7,2].Text  = "D"
Let Key[7,3].Label = "E"
Let Key[7,3].Text  = "E"
Let Key[7,4].Label = "F"
Let Key[7,4].Text  = "F"
```

```
    Let Key[7,5].Label = "%"
    Let Key[7,5].Text  = " % "
    Let Key[7,6].Label = "="
    Let Key[7,6].Text  = ""
    Let Key[8,1].Label = "y*"
    Let Key[8,1].Text  = "^"
    Let Key[8,2].Label = "x+"
    Let Key[8,2].Text  = "+"
    Let Key[8,3].Label = "?"
    Let Key[8,3].Text  = " ? "
    Let Key[8,4].Label = "?"
    Let Key[8,4].Text  = " * "
    Let Key[8,5].Label = "-"
    Let Key[8,5].Text  = " - "
    Let Key[8,6].Label = "+"
    Let Key[8,6].Text  = " + "
    Let Key[9,1].Label = "x?"
    Let Key[9,1].Text  = "?"
    Let Key[9,2].Label = "?é"
    Let Key[9,2].Text  = "CBRT "
    Let Key[9,3].Label = "M"
    Let Key[9,3].Text  = "M0"
    Let Key[9,4].Label = "M+"
    Let Key[9,4].Text  = ""
    Let Key[9,5].Label = "("
    Let Key[9,5].Text  = "("
    Let Key[9,6].Label = ")"
    Let Key[9,6].Text  = ")"
    Let Key[10,1].Label = "Mod"
    Let Key[10,1].Text  = " MOD "
    Let Key[10,2].Label = "Div"
    Let Key[10,2].Text  = " DIV "
    Let Key[10,3].Label = "Or"
    Let Key[10,3].Text  = " OR "
    Let Key[10,4].Label = "And"
    Let Key[10,4].Text  = " AND "
    Let Key[10,5].Label = "Xor"
    Let Key[10,5].Text  = " XOR "
    Let Key[10,6].Label = "Not"
    Let Key[10,6].Text  = "NOT "
    Let Key[11,1].Label = "CBit"
    Let Key[11,1].Text  = " CBIT "
    Let Key[11,2].Label = "SBit"
    Let Key[11,2].Text  = " SBIT "
    Let Key[11,3].Label = "LSL"
    Let Key[11,3].Text  = " LSL "
    Let Key[11,4].Label = "ROL"
    Let Key[11,4].Text  = " ROL "
    Let Key[11,5].Label = "Store"
    Let Key[11,5].Text  = "STM "
    Let Key[11,6].Label = "Recall"
    Let Key[11,6].Text  = "M"
    Let Key[12,1].Label = "CE"
    Let Key[12,1].Text  = ""
    Let Key[12,2].Label = "TBit"
    Let Key[12,2].Text  = " TBIT "
    Let Key[12,3].Label = "LSR"
    Let Key[12,3].Text  = " LSR "
    Let Key[12,4].Label = "ROR"
    Let Key[12,4].Text  = " ROR "
    Let Key[12,5].Label = "ASR"
    Let Key[12,5].Text  = " ASR "
    Let Key[12,6].Label = "Iconify"
    Let Key[12,6].Text  = ""
    Let Key[12,7].Label = "CH"
    Let Key[12,7].Text  = ""
    Let Dir = TheOriginDirectory
    If Not Exists("Fonts:CalcFont1")
         Dos "Copy >Nil: <Nil: " || Dir || "Fonts/CalcFont1#? Fonts:
All"
    EndIf
    Dos "Run >Nil: <Nil: c:Resident " || Dir || State.Calculator
 EndScript
```

AGA+M2

Enhanced !

Workbench 3 and Updated Compiler

Benchmark WB3 provides complete Workbench 3.0 and 3.1 support, including **AGA Support**, **AREXX**, **GadTools**, **BOOPSI**, **ASL**, **IFFParse**, and **Example Programs**.

Updated compiler features include 68020 code generation and direct support for IEEE single and double precision math.

Benchmark WB3 is available for our special introductory price of only $125. WB2 users can upgrade to WB3 for only $25.

We're offering special discounts on upgrades to Benchmark for other Modula-2 users. Call or write for our free catalog.

Money-back guarantee on all Armadillo Computing products.

Armadillo Computing

5225 Marymount Drive, Austin, Texas 78723
Phone/Fax: 01-512/926-0360 Internet: jolinger@bix.com

Turbo Pascal - Only Better

Listing 2

```
Routine "ProcessKey"
   Let X = Integer(GetWord(ObjectName,2,"_"))
   Let Y = Integer(GetWord(ObjectName,3,"_"))
   Let Class = GetWord(ObjectName,1,"_")
   If Class = "Num"
         Let X = X + 3
   ElseIf Class = "KeyR"
         Let X = X + 9
   EndIf
   If ArgCount = 1
         Do "AddKeyToDisplay",Key[X,Y].Text,Arg1
   Else
         Do "AddKeyToDisplay",Key[X,Y].Text
   EndIf
EndScript
```

✔

Please Write to:
Michael Griebling
c/o Amazing Computing
P.O. Box 2140
Fall River, MA 02722-2140

Re Color Revisited

by Dave Senger

Black as a cole mine at midnight

That's the color of WordPerfect document icons on the new Workbench screens. They look like photographic negatives. WordPerfect, and many other old programs, generate icons that just don't look good on the new Workbench screens. They need to be brought up to date. This article is about how to make your favorite Golden Oldies generate brand-new icons.

The software companies could do the job themselves, but it is unlikely that most will. They can't afford to. Take WordPerfect. They stopped development of WordPerfect for the Amiga some time ago. They generously continue to provide customer support (telephone support ended on June 30, but mail support will continue), even though the Amiga market has been a money loser for them. They could easily recompile the program with a new icon, but then they would have to maintain two versions of the latest revision to accommodate the different versions of the Workbench that are currently in use, and they would have to distribute a lot of disks to customers. I don't see that happening, so I came up with a way to solve the problem myself.

Take a look at figure 1. All of the icon images you see can be installed in WordPerfect. The one that looks like a solid black scroll is the original unrecolored WordPerfect doc icon. Next to it is a recolored version (although it is in black and white, you can tell by the additional detail). The one with my name on it is an image I recolored, then edited with a paint program. The one with 'SB' in the center is the icon used to click-start Superbase Personal. The standard Workbench Drawer icon has two images. Only the first image of dual-image icons can be installed in WordPerfect, but load files which generate dual-image icons can have them replaced with dual-image icons. The two short/wide and tall/narrow icons that say 'my WP doc' are a couple of quick and dirty icons I made to test my ideas as I was developing NewIconWP.rexx, the ARexx script I used to copy all these icon images to WordPerfect. These are only a few of the possibilities. You can edit any of these icons to suit yourself, or install the first image of any other existing legal icon whose bitplanes are small enough to fit, or make a brand-new icon that no other Amiga user has. This article includes ARexx scripts you can use to do the same thing with five other load files, and it explains how to use these scripts as models for scripts you can write yourself to do the same job on almost any other load file you have. In most cases, it isn't even hard to do.

Some programs, such as the editor and compiler from AMOS, the European BASIC programming language system, do not store within the load files themselves the icon data that they use. Instead, they simply copy separate icons which are stored elsewhere. The advantage is that these stand-alone icons can be edited or replaced without having to edit a load file. The program will use any icon that has the name, and is in the location, that it expects. The trade-off is that this makes the program slightly harder to move to another directory on your system, since you must remember to move the .info file along with it. I like this method, but most programs don't use it.

I hope you read my previous article, Re Color (*AC's TECH* Volume 3 Number 4), and kept a copy. You will need some of the information in the earlier article, especially the information on structures, to understand this one. I don't have the space to repeat that material. If you don't have the article, you can find information you will need on DiskObject, Gadget, and Image structures in Rhett Anderson's and Randy Thompson's Mapping the Amiga, published by Compute! Books, and in the Amiga ROM Kernel Reference Manual: Includes and Autodocs, Third Edition.

Editor's Note:
In the course of his article, Mr. Senger refers to several programs available through public domain and the Fred Fish Collection. In many cases, Mr. Senger refers to these programs as residing on the AC's TECH *companion disk.*

At press time, the AC's TECH *disk is still in creation and we cannot be positive all the files he has requested will find room on the disk. Although every effort will be made to supply these files in a compressed format, we must retain room for the files and programs submitted by the remaining authors in this issue. If the files are not present, we will provide an additional readme file on the disk with suggestions on where the files may be obtained.*

RecolorWP.rexx

Let's get to the good stuff right away. A number of ARexx scripts accompany this article. You will find them on the AC's TECH companion disk for this issue. The ones you decide to use need to be copied to your Rexxc directory. If you have WordPerfect, start with RecolorWP.rexx.

Assuming ARexx is correctly set up on your system (covered in the previous article), CD a Shell to the directory containing your 'wp' load file, and make sure you have an unedited backup stored safely away. Enter 'rx RecolorWP'. When offered a choice of recoloring routines, enter 1. In a few seconds your disk drive will stop

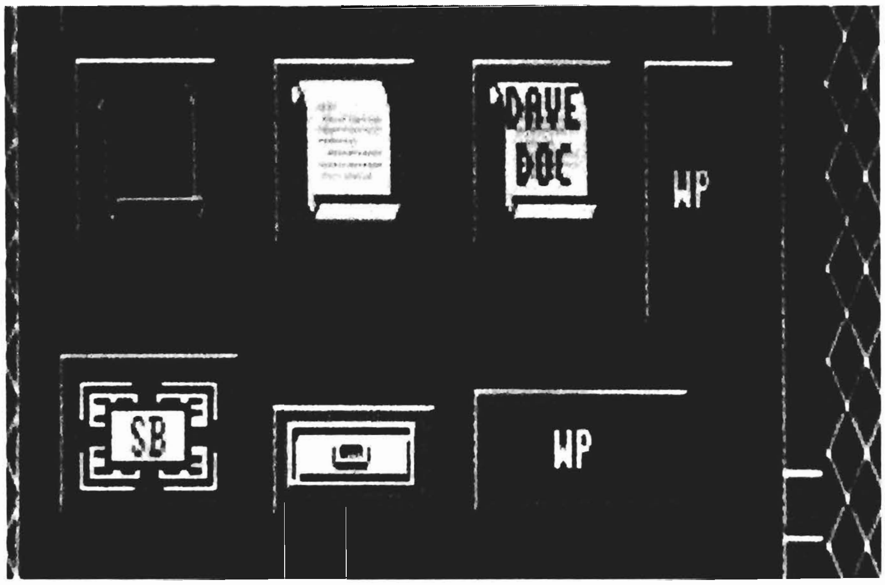


Figure 1:
All of the icon images you see can be installed in WordPerfect. The one that looks like a solid black scroll is the original unrecolored WordPerfect doc icon. Next to it is a recolored version (although it is in black and white, you can tell by the additional detail).

running, and it's all done. Run WordPerfect and save a dummy file. The icon should be correctly colored for the new Workbench screen.

RecolorWP.rexx provides sixteen different routines to recolor your WordPerfect icon. The first routine recolors the icon simply by reversing the order of the two bitplanes used to draw the image, just as RecolorIcons.rexx, from Re Color, recolors icons. It runs in two or three seconds. Three of the next fifteen routines restructure the bitplanes by doing a lot of bitwise string manipulation, and each one takes up to a minute to execute on an unaccelerated Amiga 2000. If nothing seems to be happening, don't be too quick to assume that something must be wrong. Each of these three routines processes one image line on each pass through the outer loop. These three routines are slow, but easy to understand. Each one is followed by a much faster, more subtle alternative, which does the same job. The alternatives use the BITXOR() function with masking. With all of these routines, any unused bits in each image line of each bitplane are not processed, and remain cleared (to 0), which conforms with the programming convention. For the three routines which use loops, this also speeds things up.

Starting with a fresh copy of 'wp', run the script on it again. This time, enter 2. Your edited load file should once again generate a recolored icon, exactly as it did the first time. Since the second routine takes much longer to get the same result, what is the advantage? In this case, not much. But though it is slower, the second method, which restructures the bitplanes a bit at a time, is far more flexible. You can use it to reassign all of the icon image's pixels of any color to any of the other three, which is not possible simply by swapping bitplanes. The third routine does the same job more quickly.

The fourth routine (slow) reverses the black and white colors, and also grey and blue. I used this routine to generate a recolored WordPerfect icon, which I edited with a paint program to make the icon personalized with my name, which you see in the screen shot. The fifth routine (fast) does the same job. The sixth (slow) and seventh (fast) routines rotate each of the four colors forward by one. Grey (color 0) is changed to black (1), black to white (2), white to blue (3), and blue to grey. If you run one of these routines on a wp load file four times, your icon colors will wind up right back where they started. The remaining routines reassign an icon image's colors in a variety of other ways. They are described in the list of options that the script puts on the screen.

Whatever your level of ARexx programming experience, as long as you have at least a little, you should be able to use these examples to write your own routines to reassign a program's icon image colors any way you want. The main thing to notice about the three slower routines, which do bitwise string manipulation inside loops, is that each pair of bits is edited together, since both of them determine the color register number.

Masking

The other twelve routines use masking to reassign colors. A mask is a pattern of bits that can be used to selectively operate on individual bits in a variable or string. If a bit is ANDed, or ORed, or eXclusive ORed, with another bit, the result depends on whether the second bit is a 0 or a 1. An AND operation outputs a 1 only if both input bits are 1s. If a bit pattern is ANDed with a mask consisting of 0s and 1s, only those 1s in the pattern which match 1s in the mask will appear in the output. All other output bits will be 0s. An OR operation outputs a 0 only if both input bits are 0. In the other three cases, it outputs a 1. ORing two bit patterns together produces an output which has a 1 in each location in which either or both of the original patterns has a 1. An XOR operation outputs a 1 if only one of the input bits is a 1. Otherwise, it outputs a 0. A bit pattern that is XORed with a mask will produce an output whose bits are inverted wherever they correspond with 1s in the mask, and are left unchanged wherever they correspond with 0s.

Using masking to reassign an icon image's colors takes three steps:

1) Decide which bits in the two bitplanes need to be changed.
2) Make one or more masks, as needed, to change the bits.
3) Use the mask(s) to restructure the bitplanes.

I'll explain two or three of the routines. The third one swaps the black and white colors (1 & 2). The code is:

```
c=BITXOR(a,b)
a=BITXOR(a,c)
b=BITXOR(b,c)
```

XORing two bits together produces a 1 when only one of the original bits is a 1, and a 0 in both other cases. 'a' and 'b' are strings containing Bitplane0 and Bitplane1. c=BITXOR(a,b) takes the two strings, 'a' and 'b', and produces a mask string 'c', which has a 1 bit in each position in which either 'a' or 'b' has a 1, and the other has a 0. 'c' has a 0 in all other locations. Wherever there is a 1 in 'a' and a 0 in 'b', that pixel is assigned to color register binary 01, or decimal 1, which is black on the latest Workbench screens. A 0 in 'a' and a 1 in 'b' gives binary 10, or decimal 2, which is white.

The 1s in 'c' correspond with all locations in 'a' and 'b' where pixels are colored either black or white.

Any bit eXclusive ORed with 1 will be inverted; with 0, will be left unchanged. a=BITXOR(a,c) therefore inverts all bits in 'a' corresponding with black or white pixels. b=BITXOR(b,c) does the same for 'b'. The result is that a binary 01 or a 10 in 'b' and 'a' is inverted to 10 or 01, which swaps the colors black and white.

The seventh routine rotates all four colors forward by one. Grey (0) is changed to black (1), black to white (2), white to blue (3), and blue to grey. The code is:

```
CALL MakeMaskstr()
b=BITXOR(b,a)
a=BITXOR(a,maskstr)
```

MakeMaskstr:

```
maskstr=COPIES('FFFF'x,wordWidth-1)||,
        D2C(65536-2**(wordWidth*16-width),2)
maskstr=COPIES(maskstr,height)
RETURN
```

MakeMaskstr() produces a mask string which has a 1 bit in each location where there is a used bit in a bitplane, and a 0 wherever there is an unused bit. XORing a bitplane with this mask will invert all used bits and leave all unused bits cleared.

In the line, 'b=BITXOR(b,a)', 'a' (Bitplane0) is used as a mask to restructure 'b' (Bitplane1). This line puts a 1 in 'b' in each location where 'b' has a 1 and 'a' has a 0, or vice-versa. If both bits are either 1 or 0, a 0 is placed in 'b'. a=BITXOR(a,maskstr) inverts all used bits in 'a'. So, if two used bits in 'b' and 'a' are 00, they are changed to 01 (grey to black). 01 goes to 10 (black to white), 10 goes to 11 (white to blue), and 11 goes to 00 (blue to grey).

The ninth routine swaps the grey and blue colors (0 & 3). The code is:

```
CALL MakeMaskstr()
c=BITXOR(a,b)
d=BITXOR(c,maskstr)
a=BITXOR(a,d)
b=BITXOR(b,d)
```

Notice the similarity to the third routine. c=BITXOR(a,b) produces a mask string which has a 1 bit in each location corresponding with black or white (1 or 2) in 'a' and 'b'. All other used locations in the two bitplanes correspond with either grey or blue (0 or 3). d=BITXOR(c,maskstr) inverts all the used bits in 'c', which produces a mask string containing a 1 bit in each location corresponding with either grey or blue in 'a' and 'b', and a 0 in all other locations. XORing 'b' and 'a' with this mask changes all 00 bit pairs to 11, and vice-versa, which swaps the grey and blue colors.

Now for a test. Like the ninth routine, the tenth routine reverses the grey and blue colors (0 & 3). The code is:

```
CALL MakeMaskstr()
c=BITXOR(a,maskstr)
d=BITXOR(b,maskstr)
e=BITAND(a,b)
f=BITAND(c,d)
g=BITOR(e,f)          /* g=BITXOR(e,f) also works */
a=BITXOR(a,g)
b=BITXOR(b,g)
```

In this routine, the lines 'g=BITOR(e,f)' and 'g=BITXOR(e,f)' are interchangeable. Do you see why?

I won't explain the slow routines which use loops, since they are fairly obvious, and I won't provide a fuller explanation of how the rest of the ARexx code works, since I have a lot more territory to cover. If you don't understand masking well enough to use it in your own routines, not to worry. The alternate method of doing bitwise string manipulation within loops works perfectly well. It just takes longer.

NewIconColors.rexx

Since the twelve routines which use masking are all fast, any of them can be substituted for the bitplane-swapping routine in RecolorIcons.rexx, from Re Color. This means that you have a lot of options to recolor existing icons in unusual ways. RecolorIcons.rexx can even be edited to include all of them, and provide the user with a choice of twelve options. In my script, NewIconColors.rexx, I have done most of the work for you already. This script uses a bitplane-swapping routine, plus ten of the masking routines, to provide eleven options for recoloring existing icons. Since the script's structure already exists, you can add any of your own routines you may think up, with very little work.

Another Approach

The scripts named NewIconXXX.rexx all work differently. They replace the icon images that programs generate, instead of recoloring them. You may recall from my previous article that a program generates an icon by locating in the memory it occupies all the structures and data elements it needs, and writing them to an .info file in a predetermined order. These scripts reverse the process, copying data from .info files to load files. After I had succeeded in recoloring a program's icon by swapping its bitplanes, I realized that there is a much better way. I could replace any icon data with any other data I wished, provided that there was enough room in the load file. The edited program would copy the new data to each .info file it made, resulting in a new icon image.

NewIconWP.rexx

You can use NewIconWP.rexx to get the same result that each of the first three routines in RecolorWP.rexx produces. Use an unedited wp load file to generate a document icon in the original WordPerfect colors. Recolor this icon with IconEdit; or RecolorIcons.rexx, from my previous article, and also on this issue's disk; or NewIconColors.rexx, on this issue's disk. If you use the second ARexx script, select the first recoloring option. Finally, use NewIconWP.rexx to copy the bitplanes of your recolored icon to your wp load file. Using a Shell CD'd to your WP directory, enter, for instance:

```
NewIconWP wp MyWPFile.info
```

You can also use NewIconWP.rexx to install an icon image of any size and shape in your wp load file, as long as its bitplanes will fit into the 416 bytes of space available. You can use an edited WordPerfect icon, or any other icon you have on a disk, or an icon you made yourself. If you are not sure whether an icon's bitplanes will fit, try it. NewIconWP.rexx will refuse to copy the bitplanes if they are too big. You can even copy the first image of a dual-image icon, such as the standard Workbench Drawer icon. When you click on the icon that the edited program generates, you will see the complimented image of a closed drawer, instead of the second image of an open drawer. After you have chosen a new image, RecolorWP.rexx can be used to manipulate the colors of any icon image you have installed in your wp load file.

There are four versions of WordPerfect for the Amiga. As provided, RecolorWP.rexx and NewIconWP.rexx work with 4.1.9, 4.1.11, and 4.1.12. If you happen to have the second-oldest version, 4.1.10, which I don't have, one line of each script will have to be edited, to provide the correct file length of wp 4.1.10. Instructions are in the scripts. The scripts work fine for the other three, and I am almost certain that they will also work for 4.1.10. All the editing is done within the last 800 bytes of the load file. The last 800 bytes of 4.1.9 and 4.1.11 are identical. If the last 800 bytes of 4.1.10 are identical with these two, which is almost certainly the case, the scripts will work. There is virtually no chance that WordPerfect programmers would have changed these last 800 bytes in the 4.1.10 revision, then changed them back to exactly what they had been before in the 4.1.11 revision.

If you are still using one of the older versions, it would be worth your while to write for your free 4.1.12 upgrade. Send a letter to:

Macintosh Customer Support, G100
WordPerfect Corporation
1555 N. Technology Way
OREM Utah 84057

Remember to include your WordPerfect license number.

NewIconXXX.rexx

All of the other NewIconXXX.rexx scripts can be used both of these ways, too. You can recolor original icons and copy them back to load files, or you can replace them with completely different icons. Each of these scripts copies the bitplanes of one or more icon images to a load file, provided that they are small enough to fit into the available space. If the bitplanes are too big, it will refuse to copy them to the load file. Of course, you could edit any of these scripts to make it copy bitplanes as large as you like. Don't. If you overwrite part of a load file outside of the space reserved for bitplanes, your damaged load file will almost certainly malfunction, and there is a good chance that it will bring down the system.

Each script also copies the Gadget Width (gg_Width) and Gadget Height (gg_Height), and also the Image Width(s) (ig_Width) and Image Height(s) (ig_Height) from the .info file(s) to the load file. The Image Width and Height define the size of the icon image you see on the screen. The Gadget Width and Height define the area within which a mouse click will activate the icon. Conventional programming practice is to overlay the Gadget with the Image, but to make the Gadget Height one line greater than the Image Height, so that the Gadget extends one line below the Image. Most programs follow this convention. Some do not. When I was developing these scripts, at first I just added one to the Image Height to get the Gadget Height, but this did not take into account the fact that not all icons follow the programming convention. So I edited them to make them read the Gadget Width, Gadget Height, Image Width(s), and Image Height(s) from the .info file, instead of computing the Gadget Height from the Image Height. Now the scripts copy the exact values in the .info files to the load files, whether the .info files were made following conventional programming practice or not. This ensures that the icons generated by the edited programs will look just like the icons whose images were copied.

These scripts also check the .info files to see if they use GADGHCOMP or GADGBACKFILL mode (see the previous article). Whichever mode the copied icon uses is transferred to the load file, so that the appearance of the selected icon generated by the edited program will be the same as that of the selected copied icon. If the first image of a dual-image icon is used to replace the image of a single-image icon in the load file, the default mode that the load file used originally is restored.

Any other icon data in the original load file, such as the Default Tool, is left unchanged, so that the new icon that the load file generates will behave just as the old one did. Only the appearance of the icon is changed.

To use each of these scripts, CD a Shell to the directory containing the load file you want to edit, and enter, for example:

```
rx NewIconsSpectraColorJr
```

When prompted, enter each pathname/file name that the script requests. You can also enter this data on one line, using a Shell CD'd to another directory. For instance:

rx NewIconsSpectraColorJr Paints/SpectraColorJr
Paints/MyPic.info Paints/MyBrush.info

You can use each of these scripts to attempt to copy any icon image(s) you like to your load file. If the bitplanes are too big to fit, the script will not copy them. If you would like to check the size of an icon image's bitplanes before installing them, run PrintIconData.rexx on the icon, then read the file 'FirstHexBitplanes' in RAM:.

NewIconAmigaBASIC.rexx

Since you are reading AC's TECH, you have probably had your Amiga for a while, which means that you most likely have a copy of AmigaBASIC. The original AmigaBASIC listing icon has a bitplane length of 120 bytes, but NewIconAmigaBASIC.rexx lets you install an icon image with a bitplane length of up to 240 bytes. Sounds impossible, doesn't it?

AmigaBASIC is the only program I have come across so far that allows this to be done. The reason is that the last version of the program was rather sloppily edited from the previous version. The older version used a single-image listing icon which looks like a sheet of tractor feed printer paper with lines of text on it, and the

(continued on page 34)

List of Advertisers

Company	Page Number
AMOS	CIV
Armadillo Computing	27
Comeau Computing	CII
Delphi Noetic Systems	11
Digital Imagery	3

AC's TECH Disk

Volume 4, Number 3

A few notes before you dive into the disk!

- You need a working knowledge of the AmigaDOS CLI as most of the files on the AC's TECH disk are only accessible from the CLI.

- In order to fit as much information as possible on the AC's TECH Disk, we archived many of the files, using the freely redistributable archive utility 'lharc' which is provided in the C: directory. lharc archive files have the filename extension .lzh.

  To unarchive a file *foo.lzh*, type *lharc x foo*
  For help with lharc, type *lharc ?*
  Also, files with 'lock' icons can be unarchived from the WorkBench by double-clicking the icon, and supplying a path.

AC's TECH DISK
GOES HERE!

Please notify your retailer if the AC's TECH Disk is missing.

We pride ourselves in the quality of our print and magnetic media publications. However, in the highly unlikely event of a faulty or damaged disk, please return the disk to *PiM Publications, Inc.* for a free replacement. Please return the disk to:

AC's TECH
Disk Replacement
P.O. Box 2140
Fall River, MA 02720-2140

Be Sure to Make a Backup!

CAUTION!

Due to the technical and experimental nature of some of the programs on the AC's TECH Disk, we advise the reader to use caution, especially when using experimental programs that initiate low-level disk access. The entire liability of the quality and performance of the software on the AC's TECH Disk is assumed by the purchaser. PiM Publications, Inc, their distributors, or their retailers, will not be liable for any direct, indirect, or consequential damages resulting from the use or misuse of the software on the AC's TECH Disk. (This agreement may not apply in all geographical areas.)

Although many of the individual files and directories on the AC's TECH Disk are freely redistributable, the AC's TECH Disk itself and the collection of individual files and directories on the AC's TECH Disk are copyright PiM Publications, Inc. and may not be duplicated in any way. The purchaser, however, is encouraged to make an archive/backup copy of the AC's TECH Disk.

Also, be extremely careful when working with hardware projects. Check your work twice, to avoid any damage that can happen. Also, be aware that using these projects may void the warranties of your computer equipment. PiM Publications, or any of its agents, is not responsible for any damages incurred while attempting this project.

(continued from page 31)

upper right corner folded over. You can see this icon on the right side of the BasicDemos window of the V1.2 or V1.3 Extras disk. You will have to use the bottom scroll bar to see it. The new icon looks the same, except that instead of lines of text, it has a diagram representing a flow-chart. When the programmers edited the older version, they just added the new icon image's bitplanes right after the old bitplanes, without bothering to delete them, and added some new code to overwrite some of the values in the icon's structures which had been written by the old program. Some of the old code still runs, uselessly, its work being undone by the new code. Some of it is now branched past.

Since the two sets of bitplanes are contiguous, we have 480 bytes of space in which to store two bitplanes of up to 240 bytes each. All we have to do is to reset the ig_ImageData pointer to 240 bytes less than the current value, so that it points to the start of the old set of bitplanes, instead of to the start of the new set, and store our replacement bitplanes starting from the old location. This allows an icon image much larger than the originals to be installed. Below are the lines in NewIconAmigaBASIC.rexx that reset the pointer:

```
x=SEEK('patchfile',86260,'begin')  /* Set file pointer */
x=WRITECH('patchfile',D2C(498,2))  /* Reset ig_ImageData pointer */
```

Actually, this code overwrites the second half of the data longword of the second (new) MOVE.L instruction that copies the ig_ImageData pointer to the Image structure. This longword is address-corrected when the program is loaded, adding an offset, so that the value that MOVE.L copies to the Image structure is not the value that you see (decimal 498, hex 1F2), but the actual starting address of the old bitplanes in memory. The old MOVE.L instruction copied the same value to the Image structure. In an unedited copy of AmigaBASIC, the new MOVE.L instruction writes to the ig_ImageData pointer field the starting address of the new bitplanes, which is decimal 738, or hex 2E2, plus an address-correcting offset. After editing, the two MOVE.L instructions copy the same value twice, or they would if the first was not branched past.

If you disable all seven of the following WRITECH() functions, then use NewIconAmigaBASIC.rexx on a copy of AmigaBASIC, so that only the ig_ImageData pointer is reset, the edited copy will generate the old listing icon. If you use the script as provided, you can install a recolored version of either of the original AmigaBASIC icon images, or any icon image you want, as long as its two bitplanes will fit into the 480 bytes of storage space available in the load file.

When running under OS 2.1, AmigaBASIC has some terrible bugs that can crash the system. When in the List Window, be careful not to try to move the cursor with the arrow keys or the backspace key to any position beyond the limits of the text that has been entered in the window. If no text has been entered, using any of these keys crashes my system. When in the Output Window, be careful not to enter a misspelled keyword that the program does not understand. For instance, entering 'xyz' will crash my system. AmigaBASIC is still useful for running some old programs, but I hardly use it anymore, partly because of the bugs, and partly because I have HiSoft BASIC Professional. In spite of AmigaBASIC's limitations when running under the new operating systems, it makes a good example program to use in this article because probably almost all *AC's TECH* readers have a copy, and because it has some quirks which make it more interesting than most load files.

NewIconHSBASIC.rexx and NewIconHBCompiler.rexx

If you have HiSoft BASIC Professional, V1.05, you can use these scripts to recolor or replace the icons generated by both the editor (HiSoft BASIC) and the compiler (HB.Compiler). Each load file uses one single-image icon. The editor's icon image has a bitplane length of 126 bytes, and the compiler's icon image's bitplane length is 120 bytes. To use the scripts, CD a Shell to the directory containing your editor and compiler, and enter, for instance:

```
rx NewIconHSBASIC HiSoft BASIC Myfile.info
```

or

```
rx NewIconHBCompiler HB.Compiler Myfile.info
```

NewIconsSpectraColorJr.rexx

The paint program, SpectraColorJr, generates two icons. The one it attaches to pictures and animations is a dual-image icon with a bitplane length of 184 bytes. The script will accept only a dual-image icon as a replacement. The brush icon has only one image, with a bitplane length of 96 bytes. The script lets you install either icon separately.

NewIconsFPaint.rexx

FPaint (FreePaint) is a freely distributable Deluxe Paint- style program found on Fred Fish disks #627 (Revision 37), and #548 (Revision 35). The newest version is scheduled to be on the *AC's TECH* companion disk for this issue if there is enough extra space. NewIconsFPaint.rexx works with both versions.

FPaint uses two single-image icons. The picture icon has a bitplane length of 176 bytes, and the brush icon has a bitplane length of 78 bytes. The script lets you install either icon separately.

The newest version takes into account that both the old and the new operating systems are currently in use by letting you choose to make icons for pre-Version 2 operating systems, or for the new operating systems. Have a look at the icons that the program makes, before editing the load file. You may not want to change them. This new version does the job in an interesting way. It uses only one set of DiskObject, Gadget, and Image structures for both Picture icons, and another set for both Brush icons, but a separate set of bitplanes for each icon. I haven't bothered to disassemble the program, but obviously all it does is to change the ig_ImageData pointers so that they point to the appropriate bitplanes, depending on which style of icons you choose.

It would be quite possible to edit NewIconsFPaint.rexx so that it would let you replace both sets of bitplanes for each icon, but the Image Width and Height would have to be the same for both sets, otherwise one or the other would not work. Since you can easily replace your icon images any time you want, I don't see any point in adding this feature to the script. The script replaces the old set of bitplanes for each icon. If you replace the old bitplanes for either icon with an image of a different width or height, naturally the script will edit the Gadget and Image structures for the icon, which means that the OS 2-style option that the program provides will no longer work.

If you want to edit these icons, you should know that the OS 2 Picture icon can be made from the old Picture icon by swapping the black (1) and white (2) colors, and also grey (0) and blue (3), using NewIconColors.rexx. To make the OS 2 Brush icon, do the same thing with the old Brush icon, then load your edited icon into IconEdit and replace all the blue background pixels with grey

Editing Icon Images

You may want to edit an icon image slightly, then replace it in the load file that generated it, as I did with the personalized WordPerfect document icon in the screen shot. Here's how.

Use the load file to generate an icon. Recoloring it may save some work in editing. Use NewIconColors.rexx on the icon to get the color combination you want. If you are working with WordPerfect, you can also use RecolorWP.rexx on the load file. Load the icon into IconEdit. Check the original icon's mode by pulling down the Highlight menu. The possibilities are Complement, Backfill, and Image, the last being the mode used by dual-image icons. Save the image as an IFF brush.

Start a Deluxe Paint-style program, and select a two- bitplane, four-color screen. A 640x200 screen will show the icon image the same size that it will appear on the Workbench. I prefer to use a 320x200 screen, which I find makes it a little easier to edit and cut brushes, though it distorts the image. Set the Palette to approximately the default Workbench colors, which are:

	Red	Green	Blue
Color 0 (Grey)	10	10	10
Color 1 (Black)	0	0	0
Color 2 (White)	15	15	15
Color 3 (Blue)	6	8	11

Import the IFF icon image as a brush, and stamp it down in four or five places. This makes it easy to edit several copies, and compare them. When you have an image you are satisfied with, cut it carefully as a brush. It is easy to cut a brush that is larger than the original image, since any background included in the cut brush will be invisible. If your image is too big, you may not be able to copy it back to your load file. I solved this problem by moving the brush cross hairs to the upper left corner of the image, and making a note of the x,y coordinates, then moving the cross hairs to the lower right, and checking the coordinates again. Knowing the dimensions of the image, I keep a close eye on the x,y coordinates as I cut the brush. Even so, I found it took a bit of practice to get it right. You can also find the dimensions of the image by running PrintIconData.rexx on the original icon.

Sometimes, as with the WordPerfect doc icon, you can cut an edited image that is a few pixels wider, but not taller, than the original, and still be able to install it in your load file. I will leave you to figure out why for yourself (Hint: Each image line of the WordPerfect doc icon image's bitplanes contains a few unused bits).

Save your brush as an IFF file, and import the image back into IconEdit. Once again, check the mode in the Highlight menu. Use the original mode, if that is what you want, or select the other. If you are using a dual-image icon, you will have no choice, but with a single-image icon, you have two options. Whichever mode you choose will be copied to the load file. Of course, if you are using a dual-image icon, you will have to edit two images, and load them both into IconEdit. Save your edited icon.

When you make an icon from a brush, you may want to edit the Gadget Height (gg_Height). Some programs make the Gadget Height equal to the Image Height, and others make it one greater. You may want to change it to suit yourself. You can't do it with IconEdit, but there is a way. Load the .info file into NewZAP. The Gadget Width/Height words always occupy the fourth longword of the .info file. You should see, for instance: 'E3100001 00000000 00000000 00290014'. This means that the Gadget width is hex 29 (decimal 41), and the height is hex 14 (decimal 20). The Image Width/Height words always start at either the second word of the 21st longword, or the second word of the 35th longword, depending on whether or not the .info file contains a DrawerData structure. In this case, if you wanted to increase the Gadget height by one, you would click on hex 14 and type in 15, then click on SAVE.

Now that you know how to edit these values, you may feel like experimenting. If you change the Image width or height, you will find that it does not work very well. On the other hand, you can play around with the Gadget width and height all you want.

The last step is to use the appropriate ARexx script to copy your new icon image(s) to the load file. You can use a similar method to create a brand-new icon image with a paint program, or you can make a simple icon with IconEdit. If you do not have a commercial paint program, FPaint (FreePaint), will do the job nicely. IconEdit can handle icon images no more than 80 pixels wide by 40 lines high. If you want to work with larger images, you will need to get another icon editor.

Finding and Editing Structures and Bitplanes in Load Files

Structures found in .info files are always complete. In other words, an .info file structure always contains all the data it needs in each of its fields. For instance, an .info file Image structure will always contain an ig_Width word which says how many pixels wide the image is. It sounds impossible, but this is not always the case with load files, even though all of the data in an .info file is copied from the memory occupied by the program that generates it.

The reason for this is that even though load files are popularly called programs, they are really not quite the same things. Most of the time, the distinction hardly matters, but in this case, it does. Since the Amiga is a multitasking computer, it is impossible for a programmer to know what memory addresses will be available when a program is loaded. That depends on whether any programs and data have been loaded already. This means that Amiga programs must be relocatable, or capable of being loaded into many different areas of memory. An Assembly Language program consists of some combination of instructions, data, and storage space. An Assembly Language source file is assembled and linked to produce a load file. This file contains the instruction, data, and storage elements of an Assembly Language program; plus data which tells the AmigaDOS program loader exactly what is in the load file, and where its various parts can be found; plus data that the program loader uses to correct any of the program's addresses that need to be updated, after the loader decides exactly where in memory to install the program. These last two types of data are used by the program loader, then thrown away. Only the instructions, data, and storage areas of the program, with its addresses corrected, are actually installed in memory, and only these elements form the program that runs on the computer.

An Assembly Language programmer can handle structures in three ways that I know of. The simplest is to include the structure in the source file, with its elements correctly entered, usually in a DATA section, though anything can be put into a CODE section. This is how it is done in WordPerfect. In this case, all you have to do is find the structure in the load file to be able to edit the various elements, such as ig_Width, that it contains.

A second way is to include the space occupied by the structure in the source file, but to let the program write the various elements to the structure after it starts running. In this case, it is harder to edit the structure's elements. If you find the space occupied by the structure in the load file, and write in the values you want, they will be overwritten after the program starts running. Instead, you must find the instructions that write data into the structure, and edit the data that the instructions write. Then, when the program runs, these instructions will write the data that you want into your structure.

A third way is not to include even the space for the structure in your source file. Instead, you can have the program allocate memory for the structure after it starts running, then write all the necessary values into the structure. In this case, you will not be able to find even the area occupied by the structure in the load file. AmigaBASIC handles things this way, copying the Image structure data to a stack frame, and the DiskObject structure (with its embedded Gadget structure) data to a block of allocated memory. In this case also, you must find the instructions that write data into the structures, and edit the data that the instructions write.

Not only can structures make life difficult, even the image data can be a problem. In the great majority of cases, the image data in an .info file will be identical to the image data in the load file of the program that generated the .info file. In these cases, it is very easy to find the image data in the load file. But in a very few cases, a program will process its image data after it starts running, to produce the bitplanes that it copies to .info files. In these few cases, your job will be harder. FindIconData.rexx will not find the image data, since it searches for an exact match for the bitplanes it copied from an .info file generated by the program. Since these bitplanes have been processed from the image data in the load file after the program started running, FindIconData.rexx will not find a match.

What this boils down to is that no single, simple method exists that will always locate within a load file the various data elements that need to be edited. My script, FindIconData.rexx, will do all the work for you in many cases, and even when it cannot find the structures you need to edit, it will almost always find at least the image data. But when you are working with a load file such as AmigaBASIC or SARGON_III, a disassembler is the only tool I know of that will always find everything you need.

FindIconData.rexx

Some of what I just said sounded a little discouraging. The good news is that in many cases, the job will be easy. If you want to replace the icon(s) in a load file for which I have not provided a script, you need to locate within the load file the various pieces of data that need to be edited, then write an ARexx script to do the job.

Run your original program, and save one copy of each icon that it generates. CD a Shell to the directory containing your load file, then enter, for instance: 'rx FindIconData MyProg'. When prompted, enter the pathname/file name of one of the .info files that you just made. FindIconData.rexx will search the load file and print the number of bytes that must be counted from the start of the file to arrive at each possible instance of Gadget or Image structure data elements, or bitplanes. It does this by searching for a six-byte string from each Image structure, for a ten-byte string from the Gadget structure, and for each set of bitplanes. If there are two Image structures, the two Image structure search strings will almost always be identical, so each one will be located twice. It will be up to you to figure out which Image structure belongs with which set of bitplanes. If the program generates two icons of the same size and shape, even more Image structures may be located, not to mention two Gadget structures. Again, you will have to sort out which is which. This will be unusual, however. Make notes of the data you need, and repeat the process for each icon that the program generates.

If the script generates so much data that it threatens to scroll off the top of the screen, tap the space bar to pause, and hit <BACK-SPACE> to resume. If you do not want to write down the data you need, use redirection to send the script's output to a file in RAM:. Enter, for instance:

```
rx >RAM:Temp FindIconData MyProg MyFile.info
```

This will produce a temporary file in RAM: that you can read or print. To send the output directly to the printer, replace '>RAM:Temp' with '>PRT:'.

The script will turn up a few false locations, especially when using a six-byte string to search for Image structures. The same byte pattern may occur elsewhere. The shorter a search string, the less likely it is to be unique. When the script finds a data string, such as the Width/Height/Depth of an Image structure, the location number it prints will be the value to use in your script's SEEK() function, before writing the replacements from the .info file to the load file.

You may want to edit the search strings used in FindIconData.rexx. If you do, don't include any non-null address pointers in your search strings. When the program loader uses a load file to install a program in memory, it corrects these address pointers. Then, when the program copies the structures containing these pointers to an .info file, they will no longer have the same values that they had in the load file.

FindIconData.rexx finds structures in load files by looking for exact matches for strings of data it copies from .info files, so it will never find a match for a string containing a non-null address pointer whose value has been changed. Null address pointers can be used, since they are not corrected.

ARexx does not have any functions which can directly search files for a pattern of bytes, but it has some powerful string- search functions. ARexx strings may not exceed 65535 bytes, but many load files are much longer. So, FindIconData.rexx divides a load file into one or more strings, each one holding 65535 bytes or less, and searches each string for the icon structures' data and bitplane data. It is possible that a piece of data being searched for will be located on either side of the boundary between two strings, in which case it will not be found, since the search function will not find a complete match in either string. The script solves this problem by making substrings consisting of 4000 bytes from the end of one string, plus up to 4000 bytes from the beginning of the next, and searching the substrings, too. That way, nothing is missed.

A side effect of this design is that the same data element may be located twice, if it lies within a substring, but not on the boundary between two larger strings. That is what will happen if you use FindIconData.rexx to find the icon data in the version of FPaint (FreePaint) on this issue's disk. This happens only occasionally, and even when it does, it is a minor nuisance.

In the simplest cases, with a load file such as WordPerfect's wp, which contains data for only one single-image icon, FindIconData.rexx will print only one possible location for the Gadget structure data, the Image structure data, and the bitplanes. It never hurts to check, but in such cases, it is unlikely that the locations are false. When the script prints two or more locations for the same data item, no more than one can be correct, and you will have to find out which one that is.

To do this, I use NewZAP 3.3, an AmigaDOS 2-compatible disk file editor from Fred Fish disk #574. This binary file editor displays files in 512-byte sectors, in both hex and ASCII. It has a fast and easy search function, and you can print sectors right from the program. With a bit of arithmetic, you can use the location numbers printed by FindIconData.rexx to find the sector, and the location within it, of the data that the script found. With some practice, you will be able to tell from the context whether this is the data that you want, or not.

It is possible to do the same thing by TYPE HEXing the load file to a temporary file, and reading it with a file reader such as More. Don't bother. NewZAP is much less trouble to use, and it also lets you do small edits directly, which is sometimes very handy.

Deciding from the context whether or not you have found data that is part of a Gadget or Image structure is not as hard as you might think. You just need to know what those structures typically look like. Take a look at the Gadget and Image structure definitions in my previous article, Re Color.

gg_NextGadget should be 00000000.
gg_Flags should be 0004, 0005, or 0006.
gg_Activation should be 0003.
gg_GadgetType should be 0001.
ig_Depth should be 0002.
ig_PlanePick should be 03.
ig_PlaneOnOff should be 00.
ig_NextImage should be 00000000.

All values are in hex, as they will appear in NewZAP.
The Gadget Width/Height and the Image Width/Height are sometimes identical. Here is an easy way to tell the difference between the two structures.

gg_Width and gg_Height are followed by gg_Flags.
ig_Width and ig_Height are followed by ig_Depth.

If the width/height words are followed by 0004, 0005, or 0006, the value is gg_Flags, and you have a Gadget structure. If they are followed by 0002, the value is ig_Depth, and you have an Image structure.

Once in a while, you may even find structures that are not used at all. If you run FindIconData.rexx on HB.Compiler, from HiSoft BASIC Professional, Version 1.05, you will find two sets of DiskObject, Gadget, and Image structures. FindIconData.rexx finds a gg_Width/gg_Height/etc. string from a Gadget structure, and does not search for any DiskObject structure elements as such, but remember that the Gadget structure whose elements it searches for is embedded in a DiskObject structure.

If you disassemble the load file, you will see that ReSource does not attach labels to any of the structures, or their elements, in the first set. This does not guarantee that this first set of structures is not used, since it is possible to access anything in a program by means of an offset from an address register, without using a label. You will also see that the second set of structures and the bitplanes are close together. That does not prove that the second set contains the structures that are used. Still, both of these observations taken together are very suggestive, and it turns out that the second set of structures in fact contains the elements that need to be edited. My guess is that the programmers simply forgot that they had put in the first set of structures when they added the second set and the image data.

For those really intractable cases, when FindIconData.rexx is not able to find all of the data that you want, you need a disassembler. To use one, you need to know some Assembly Language. Not necessarily a lot, but at least a little. In case you do not have one, you will find Version 5.12 of ReSourceDemo on Fred Fish disk #852. This is the latest version available. It works very well, though it is not always able to distinguish between code and data on its own. You will need at least 2MB of RAM to disassemble the larger load files. I have tried a couple of other freely distributable disassemblers, but I am not going to tell you about them. The experience was very painful, and I can't talk about it yet.

If FindIconData.rexx locates the bitplanes, but cannot find the other data you need, it means that the program copies the data to the Gadget and Image structures after it starts running. When using NewZAP or ReSourceDemo to locate the data, keep in mind that programmers often group the icon structures and the image data close together. The Programming Laws of the Universe don't compel them to, but usually this is the most convenient way to do the job. Check the area two or three hundred bytes on either side of the bitplanes, and in the majority of cases, you will find your structures. It is just possible that a program will allocate memory for structures after it starts running, then write the various elements to the structures, in which case you will not find the structures anywhere in the load file. In this very unusual situation, you will need enough skill with a disassembler to locate the instructions that copy data to the structures. Sometimes, the quickest way is to search the program for the 'magic' number, hex E310, which appears at the start of every valid DiskObject structure. You may find this word in several locations, but it is almost sure to appear once either at the start of the DiskObject structure, or in the data part of a MOVE instruction that copies this value to the DiskObject structure. A programmer may also copy this value to a data register, then MOVE.W the lower half of the data register to the start of the DiskObject structure. He is most likely to do it this way if he has more than one DiskObject structure to fill in. Once you have found the MOVE instruction that copies this value, the other MOVE instructions you need to locate will almost always be close by.

Help is on the way

If you want to write some of these NewIconXXX.rexx scripts yourself, you will find it easier if you install some software on your hard disk. Here is what I did on my system.

I copied the entire ReSourceDemo drawer to my hard disk. If you click open the drawer and double-click on the ReadMe icon, you will get some instructions. The key information is copied below:

For a permanent installation, just copy the above files as shown below:

```
copy RSDemo                    <anywhere in your path, e.g. 'C:'>
copy RSDemo.info                   <as above>
copy libs/ReSourceloader.library  libs:  ;or libs:ReSource/
copy libs/ReSourcemenus.library   libs:  ;or libs:ReSource/
copy libs/ReSourcesyms.library    libs:  ;or libs:ReSource/
copy libs/ReSourcehelp.library    libs:  ;or libs:ReSource/
copy s/RS.keytable             s:
copy s/RS.macros               s:
```

RSDemo needs the four libraries to work.

The simplest way to use ReSourceDemo's functions is to select them from the menus. You can get faster results by using keyboard shortcuts, which you can program yourself. RS.keytable provides dozens of ready-made key combinations to get you started. RS.macros does the same for macros, which you can also add yourself. Unfortunately, you can't save either of them, or any other files, in the demo version.

I have included an Extras drawer in the ReSourceDemo drawer. The macros file provided with version 5.12 is very small, so I have added an old RS.macros file. I don't remember which version it is from. It includes a macro which will disassemble the boot sector of a bootable disk in DF0:, which will give you a small, but very impressive, demonstration of what ReSource can do with a macro. To use this file instead of the V5.12 file, just copy it to your s directory, and run RSDemo.

The keyboard shortcuts are useful only if you know what they are. I recommend the program, 'ShowKeys', from V3.06 of ReSourceDemo, on Fred Fish disk #232. Enter 'ShowKeys' in a Shell to see the key bindings printed out on your screen. To get a hard copy, enter 'ShowKeys >PRT:'. This program will print most of the key bindings, but not quite all of them. A very important one it leaves out is the multiply (*) key on the upper right corner of the numeric keypad. Use this key to call up the label requester. Making meaningful labels is one of the main jobs in disassembling a program, so you will use this key a lot.

The documentation provided with V5.12 is quite limited, so I have attempted to included two copies of the manual from V3.06. One copy is the original. The second is a WordPerfect file. In the second copy, I have cleaned up the formatting and added page numbers. You will need WordPerfect to print it. I prepared this copy for my own use, but you might as well get the benefit of it. This manual is outdated, and does not provide information about things that have been added, such as the on-line Hyper-help feature, but it contains a lot of useful information. You can get all of this material from previous releases of ReSourceDemo, so I don't think anyone should mind.

The last item in the Extras drawer is arp.library, version 39.1. ReSourceDemo requires version 34 or higher of arp.library to make file requesters. If you don't have it already, copy this file to your Libs directory. If you have arp.library, and would like to see which version you have, CD a Shell to your Libs directory, and enter 'version arp.library'.

In the main drawer, you will find version 3.3 of NewZAP, the latest, from FF #574. I just copied the entire drawer to my hard disk.

The last item is HexCalc, from Fred Fish disk #67, on the AC's TECH disk. This is a small, handy, Hewlett-Packard style base converter and calculator. The load file itself is called HEX. Your c directory might be a good place to put it. Then, you will be able to make the calculator pop up on your Workbench screen just by entering 'run HEX' in a Shell.

Using the Tools

Got it all set up?
Okay, lets take it for a quick spin. We will use the software to find the data we need in AmigaBASIC to write an ARexx script that will install a new icon image.

FindIconData.rexx will not find the Gadget and Image structures in AmigaBASIC, because this program copies the various elements to the structures only after it starts running. If you use an icon made with the last revision of AmigaBASIC, the script will locate the new bitplanes, but it will not tell you about the old ones. If you relied on this script, you would never know that the load file has enough room to store an icon image whose bitplanes are twice as big.

Load an unedited copy of AmigaBASIC into NewZAP. Since FindIconData.rexx cannot find the Gadget and Image structures, we will look for the instructions that write data to them. The 'magic' number, hex E310, must be placed at the start of the DiskObject structure sometime before the program makes an icon (an .info file) and copies it to disk. If the programmer did not do this when he wrote the source file, then a MOVE instruction must do it when the program runs. Select String Search from the Search menu, and enter '$E310' in the requester, without the quotes. When NewZAP finds the first three occurrences, select Continue to find the last occurrence in the load file, in sector 169. You will see the longword, hex 30BCE310. This is the MOVE.W instruction that copies hex E310 to the start of the DiskObject structure. Count back six or seven longwords, until you find 2D7C0000 02E2. This is the MOVE.L instruction that copies the new ig_ImageData pointer (after it has been address- corrected when the program is loaded) to the Image structure. If you count back about another five longwords, you will find 2D7C 000001F2. This is the MOVE.L instruction left over from the previous revision that copied the old ig_ImageData pointer to the Image structure. Hex 2E2 - 1F2 = F0, or decimal 240, the number of bytes in one pair of bitplanes.

So far, so good. Now, click NewZAP to the back of the Workbench screen, and start RSDemo (ReSourceDemo). Use <Right Amiga>-o to bring up the file requester, and load AmigaBASIC. RSDemo will take about 20 seconds to allocate memory and do the preliminary disassembly. Select 'Disassemble' from the (P)ROJECT menu. The program will take about another 20 seconds to do as much of the rest of the disassembly as it can on its own. Notice that the title bar displays the byte position of the top line (the cursor line) of the program. Tap F7 to bring up the search string requester, and enter 'E310' without the quotes. Unlike NewZAP's search function, RSDemo's is case-sensitive. Tap F9 to start the search. You will see the instruction 'MOVE.W

#$E310,(A0)', which copies the 'magic' number to the start of the DiskObject structure. Four instructions above, you will see 'MOVE.L

#lbL0146EE,(-14,A6)', which is the instruction added in the new revision of AmigaBASIC to copy the new ig_ImageData pointer to the Image structure. Four instructions above that, you will find 'MOVE.L

#lbL0145FE,(-14,A6)', which is the instruction which previously copied the starting address of the old bitplanes to the Image structure. Seven instructions above that, you will see 'MOVE.L

A0,(-14,A6)'.

This instruction, which was intended to copy an old ig_ImageData pointer to the Image structure, may be a relic left over from an even earlier incarnation of AmigaBASIC. I think that this instruction is a programmer's error. It copies the address of the Image structure itself (gg_GadgetRender) to the ig_ImageData field in the Image structure, which ought to hold the pointer to the bitplanes. Notice that the MOVE.W instruction at 014320, which copies the value 0 to ig_LeftEdge, at the start of the Image structure, uses the same address as that MOVE.L instruction, which gets its address from the LEA (Load Effective Address) instruction that precedes it. Since ig_LeftEdge is the first element in the Image structure, of course its address is the same as gg_GadgetRender, the address of the Image structure. Also, notice that at 0143AA there is a LEA instruction which is almost identical to the one at 01433A. Immediately after the second LEA instruction, the loaded address is copied to the gg_GadgetRender field in the Gadget structure. Since the incorrect address that the MOVE.L instruction at 01433E copies to the ig_ImageData field is overwritten by the newest MOVE.L instruction, it does no harm. Notice that the target address, (-14,A6), is the same in all three cases.

By the way, AmigaBASIC copies the Image structure data to a stack frame created for the purpose a few instructions before, and the data for the DiskObject structure and its embedded Gadget structure are copied to a 78-byte block of memory also just previously allocated, so not only do none of these structures exist in the load file, neither does even the space that they occupy. Check the code from 0142D0 onwards, to see the stack frame and memory allocations.

ReSource generates automatic labels using the Offset from Start (the number of bytes from the start of the program) to make each label unique. Select Offsets from the first Show... submenu of the (O)PTIONS menu. The Offset from Start for each line will appear in hex in the left column. Scroll down to 0145FE, using the <Down Arrow> key, and you will come to the start of the old set of bitplanes. Go a little farther to 0146EE, and you will see the start of the new bitplanes. Subtract hex 0145FE from 0146EE, and you will see that the difference is hex F0, or decimal 240 bytes. Scroll back to the code you left. Now, set the data type to longwords with <Left Amiga>-L. You will see the code in hex, except for the labels. It exactly matches the hex code that you see in NewZAP. Some other keyboard shortcuts are <Left Amiga>-C, -A, -B, and -W, to change the data type to code, ASCII, bytes, or words. You can also temporarily change the data type by holding down the left mouse button plus one or more keys. The left mouse button must be used with all of the following combinations:
+<Left Alt> displays the data as ASCII.
+<Left Alt>+<Left Shift> displays words.
+<Left Alt>+<Left Shift>+<Ctrl> displays longwords.
+<Left Shift> displays bytes.

A load file contains a lot of data in addition to the program's instructions, data, and storage space. When ReSource loads a load file (see the (P)ROJECT menu), it strips all of this extra data, so that the Offset from Start displayed in the title bar or the left column will always be substantially less than the byte position of the same item in a load file displayed by NewZAP.

ReSource multitasks, so if you have enough RAM, start a second copy of RSDemo, select 'Open binary file' from the (P)ROJECT menu, and load AmigaBASIC again. Loading a load file this way retains all of the data that would otherwise be stripped out, so that the offsets will be the same in ReSource as in NewZAP. The ReSource offsets will be in hex, so you will have to convert them to decimal, then calculate the NewZAP sector (512 bytes each), and the position within it, to find the item you have located in ReSource. HEX will do this for you painlessly. You can also use ARexx to convert offsets from hex to decimal, or from decimal to hex. Click into a Shell, and enter, for example:

```
    rx "SAY X2D(FFFF)"
```

or

```
    rx "SAY D2X(65535)"
```

Once again, switch on the Offsets in the left column, and search for 'E310'. This will find the code that you found already in the first copy of RSDemo, but this time, you will not have labels, so when you set the data type to longwords, you will see the hex data exactly as it appears in NewZAP. Move any instruction or data item you would like to locate in NewZAP to the cursor line (the top line), then change the data type. You will see the exact hex data that you need to find, along with an offset value you can convert to decimal and use to locate your data in NewZAP. You can also find what you want in NewZAP by entering the hex data that you see in RSDemo into the NewZAP search string requester. Short strings will often appear in more than one place in a load file, but strings of ten to fourteen bytes will almost always be unique. You can also use the offsets to calculate the values to use in your ARexx script SEEK() functions, before writing new data to the load file.

Use <Left Amiga>-M to switch back and forth between the two copies of ReSourceDemo, and NewZAP. You can also pull down the ReSourceDemo screens.

➛

Below is a portion of the disassembly. I added the comment lines myself. You will notice that some of the code is duplicated, to no apparent purpose. That is what the disassembler says - don't blame me. When you rummage through load files with a disassembler, you will often come across bits and pieces of junk that just never got edited out:

```
lbC01431C MOVEQ    #0,D0
01431E    MOVEQ    #0,D1
014320    MOVE.W   D1,(-$18,A6)    ;ig_LeftEdge (0)
014324    MOVE.W   D1,(-$16,A6)    ;ig_TopEdge (0)
014328    MOVE.W   #$29,(-$14,A6)  ;ig_Width (dec. 41)
01432E    MOVE.W   #$14,(-$12,A6)  ;ig_Height (dec. 20)
014334    MOVE.W   #2,(-$10,A6)    ;ig_Depth
01433A    LEA      (-$18,A6),A0
01433E    MOVE.L   A0,(-14,A6)     ;ig_ImageData???
014342    MOVE.B   #3,(-10,A6)     ;ig_PlanePick
014348    CLR.B    (-9,A6)         ;ig_PlaneOnOff (0)
01434C    MOVE.L   D0,(-8,A6)      ;ig_NextImage (NULL)
014350    MOVE.L   D0,(-$1C,A6)
014354    TST.W    (14,A6)
014358    BEQ.B    lbC01436C  ;Branch past next 4 inst's
01435A    MOVE.L   #lbL0145FE,(-14,A6) ;ig_ImageData
014362    MOVEA.L  (-4,A6),A0 ;DiskObject Struct. Add. in A0
014366    MOVE.L   D0,($32,A0) ;do_DefaultTool (NULL)
01436A    BRA.B    lbC014380

lbC01436C MOVE.L   #lbL0146EE,(-14,A6) ;ig_ImageData
014374    MOVEA.L  (-4,A6),A0 ;DiskObject Struct. Add. in A0
014378    MOVE.L   #AmigaBASIC.MSG,($32,A0) ;do_DefaultTool
(:AmigaBASIC)
lbC014380 MOVEA.L  (-4,A6),A0 ;DiskObject Struct. Add. in A0
014384    MOVE.W   #$E310,(A0)       ;do_Magic
014388    MOVEQ    #1,D0
01438A    MOVE.W   D0,(2,A0)         ;do_Version
01438E    MOVE.W   #$29,(12,A0)      ;gg_Width (dec. 41)
014394    MOVE.W   #$14,(14,A0)      ;gg_Height (dec. 20)
01439A    MOVE.W   #5,($10,A0) ;gg_Flags
(GADGIMAGE+GADGBACKFILL)
0143A0    MOVE.W   #3,($12,A0)       ;gg_Activation
0143A6    MOVE.W   D0,($14,A0)       ;gg_GadgetType
0143AA    LEA      (-$18,A6),A1
0143AE    MOVE.L   A1,($16,A0)       ;gg_GadgetRender
0143B2    MOVE.B   #4,($30,A0)       ;do_Type
0143B8    SUBA.L   A1,A1
0143BA    MOVE.L   A1,($36,A0)       ;do_ToolTypes (NULL)
0143BE    MOVE.L   #$80000000,D0     ;NO_ICON_POSITION
0143C4    MOVE.L   D0,($3A,A0)       ;do_CurrentX
0143C8    MOVE.L   D0,($3E,A0)       ;do_CurrentY
0143CC    MOVE.L   A1,($42,A0)       ;do_DrawerData (NULL)
0143D0    MOVE.L   A1,($46,A0)       ;do_ToolWindow (NULL)
0143D4    MOVE.L   #$1000,($4A,A0)   ;do_StackSize
0143DC    MOVE.L   A0,-(SP)          ;DiskObject Struct. Add.
0143DE    MOVE.L   (8,A6),-(SP)      ;File Name Address
0143E2    JSR      (lbC0178E4).L     ;PutDiskObject STUB
```

The disassembled line at 0143BE copies the NO_ICON_POSITION system constant to D0 (data register 0), which is then copied to do_CurrentX and do_CurrentY. This value tells the Workbench that the newly made icon has not yet been assigned a permanent position, and to find a location for it in the window that is not occupied by another icon. If you TYPE HEX an icon .info file that has just been made with AmigaBASIC, you will see the value hex 80000000 at 003A, and at 003E. If you Snapshot the icon, then TYPE HEX the .info file again, you will see that the values have been replaced with the current position of the icon.

The last three lines push the DiskObject structure address and the file name address onto the stack, then jump to the PutDiskObject stub. PutDiskObject is the function that creates an icon by copying whatever data are needed from the memory occupied by the program to an .info file. Most Amiga library functions, such as the PutDiskObject function from icon.library, are written to expect 'arguments', such as the DiskObject structure address, to be in specific registers when the function is called. In Assembly Language programming, this is the programmer's responsibility. In C programming, with some compilers, the convention is to push these arguments onto the stack. A stub (STack sUBroutine) is a small subroutine which copies arguments from the stack to the correct registers, then calls the library function.

Below are two sequences of the load file, TYPE HEXed, that represent the code disassembled above, followed by the image data. I left out a big chunk of data between the two sequences. The six underlined and bolded words are all edited by NewIconAmigaBASIC.rexx. In order, they are: Image width (ig_Width), Image height (ig_Height), the second half of the pointer to the start of the bitplanes (ig_ImageData), Gadget width (gg_Width), Gadget height (gg_Height), and Gadget flags (gg_Flags). Naturally, their order below corresponds with their order in the disassembly above. NewIconAmigaBASIC.rexx edits them in a different order.

If you convert the byte positions of these words from hex to decimal, you will get the values used in the SEEK() functions in NewIconAmigaBASIC.rexx. The c directory's TYPE HEX command counts the initial byte in its output as byte 0. The ARexx SEEK() function counts the initial byte in a file as byte 1. If you CALL SEEK('filelabel',1,'begin'), the file pointer will be positioned just after the initial byte in the file, so that a CALL WRITECH('filelabel',string) will overwrite the file with the string starting with the second byte of the file, which TYPE HEX numbers as byte 1.

```
 150A0: 70007200 3D41FFE8 3D41FFEA 3D7C0029
p.r.=A.è=A.ê=|.)
150B0: FFEC3D7C 0014FFEE 3D7C0002 FFF041EE   .ì=|...î=|...?Aî
150C0: FFE82D48 FFF21D7C 0003FFF6 422EFFF7   .è-H.ò.|...öB..?
150D0: 2D40FFF8 2D40FFE4 4A6E000E 67122D7C   -@.ø-@.äJn..g.-|
150E0: 000001F2 FFF2206E FFFC2140 00326014   ...ò.ò n.ü!@.2'.
150F0: 2D7C0000 02E2FFF2 206EFFFC 217C0000   -|...â.ò n.ü!|..
15100: 03DF0032 206EFFFC 30BCE310 70013140   .ß.2 n.ü0?ã.p.1@
15110: 0002317C 0029000C 317C0014 000E317C   ..1|.)..1|....1|
15120: 00050010 317C0003 00123140 001443EE   ....1|....1@..Cî
15130: FFE82149 0016117C 00040030 93C92149   .è!I...|...0.É!I
15140: 0036203C 80000000 2140003A 2140003E   .6 <....!@.:!@.>
15150: 21490042 21490046 217C0000 1000004A   !I.B!I.F!|.....J
15160: 2F082F2E 00084EB9 00000000 508F4A80   /./...N†....P.J.
     -----
     -----
     -----
```

```
15CA0: 02020000 61756469 6F2E6465 76696365   ....audio.device
15CB0: 00000000 00000000 19FFFFE6 00001FFF   ..........æ....
15CC0: FFE78000 19FFFFE7 E0001FFF FFE7F800   .ç.....çà....çø.
15CD0: 19FFFFE0 00001FFF FFFFFE00 19FFFFFF   ...à......?.....
15CE0: E6001FFF FFFFFE00 19FFFFFF E6001FFF   æ.....?.....æ...
15CF0: FFFFFE00 19FFFFFF E6001FFF FFFFFE00   ..?.....æ.....?.
15D00: 19FFFFFF E6001FFF FFFFFE00 19FFFFFF   ....æ.....?.....
15D10: E6001FFF FFFFFE00 19FFFFFF E6001FFF   æ.....?.....æ...
15D20: FFFFFE00 00000000 00007FFF FFFE0000   ..?..........?..
15D30: 66000019 8000607F FFF86000 66000018   f.....'..ø'.f...
15D40: 1800607F FFF80600 6600001F FF80607F   ..'..ø..f.....'.
15D50: FFFF8180 66000000 1980607F FFFF8180   ....f.....'.....
15D60: 66000000 1980607F FFFF8180 66000000   f.....'.....f...
15D70: 1980607F FFFF8180 66000000 1980607F   ..'.....f.....'.
15D80: FFFF8180 66000000 1980607F FFFF8180   ....f.....'.....
15D90: 66000000 19806000 00000180 7FFFFFFF   f.....'.........
15DA0: FF800000 00000000 19FFFFE6 00001FFF   ..........æ....
15DB0: FFE78000 19FFFFE7 E0001FFF FFE7F800   .ç.....çà....çø.
15DC0: 19FFFFE0 00001FFF FFFFFE00 19FFFFFF   ...à......?.....
15DD0: E6001FFF FFFFFE00 19FFFFFF E6001FFF   æ.....?.....æ...
15DE0: FFFFFE00 19FFFFFF E6001FFF FFFFFE00   ..?.....æ.....?.
15DF0: 19FFFFFF E6001FFF FFFFFE00 19FFFFFF   ....æ.....?.....
15E00: E6001FFF FFFFFE00 19FFFFFF E6001FFF   æ.....?.....æ...
15E10: FFFFFE00 00000000 00007FFF FFFE0000   ..?..........?..
15E20: 66300019 8000603F FFF86000 66000C18   f0....'?.ø'.f...
15E30: 18006000 4C980600 66003F1F FF806000   ..'.L...f.?...'.
15E40: 0C018180 66000C01 99806000 33018180   ....f.....'.3...
15E50: 6600C0C1 99806003 003F8180 6600C0C0   f.À?..'..?..f.ÀÀ
15E60: 19806000 33000180 66000C00 19806000   ..'.3...f.....'.
15E70: 4C800180 66003F00 19806000 0C000180   L...f.?...'.....
15E80: 663FFFFF 19806030 00030180 7FFFFFFF   f?....'0........
15E90: FF806963 6F6E2E6C 69627261 7279003A   ..icon.library.:
15EA0: 416D6967 61424153 49430000 000003EC   AmigaBASIC.....ì
```

The hex code from 150A0 to 1506B coincides with the disassembly. Hex 70007200 is the machine code for the two MOVE Quick instructions at the start of the disassembly. Hex 4EB9 is the last Jump to SubRoutine. The following longword is null. After address-correction on loading, this longword will contain the jump address. The Offsets from Start for the disassemblies and the hex code are all printed in hex. Notice that the TYPE HEX offsets are much higher numbers than the ones displayed by the first copy of RSDemo, but coincide with those in the second copy. The difference is due to the fact that the first copy of the disassembler stripped much of the load file data, displaying only the program's code, data, and storage space.

The image data (480 bytes, two sets of two bitplanes), begins at 15CB2, with 0000 00000000 19FFFFE6. It ends with FF80, the two bytes at 15E90.

PrintIconData.rexx

If you run PrintIconData.rexx on a single-image icon .info file, the script will deposit three files in RAM:. They are: FirstIconImage, FirstBitplanes, and FirstHexBitplanes. With a dual-image icon, you will get six files, the last three beginning with 'Second' instead of 'First'. FirstIconImage is a pseudo- image of the first icon image. The script computes a decimal digit from 0 to 3, which is the color register number, for each pixel. It substitutes a unique character for each digit, then writes each line of characters, producing a file you can read with More or Ed, displaying the image. You can make PrintIconData.rexx create a pseudo-image using the digits 0 - 3, by disabling this line in the internal function Process2:

```
g=TRANSLATE(g,outtable,XRANGE('0','3'))
```

The disadvantage is that the digits are so similar that they produce very little contrast, making it hard to see the image. Using substitute characters makes it easier to see the image, but there is no ideal choice of characters for all images. If you are not satisfied with the contrast, try enabling a different code line from the following lines at the start of the script:

```
/* outtable=X2C(2B)||X2C(23)||X2C(B7)||X2C(3A) */
/* outtable=X2C(2B)||X2C(23)||X2C(7E)||X2C(A6) */
/* outtable=X2C(2B)||X2C(23)||X2C(3D)||X2C(A7) */
/* outtable=X2C(2B)||X2C(23)||X2C(3D)||X2C(4F) */
/* outtable=X2C(B7)||X2C(23)||X2C(3D)||X2C(4F) */
outtable=X2C(B7)||X2C(23)||X2C(D7)||X2C(4F)
```

FirstBitplanes writes the bits for the first image line of Bitplane0, followed by the corresponding bits from Bitplane1. The pairs of lines are written in order for the whole image. In both FirstBitplanes and FirstIconImage, any unused bits in each image line of the bitplanes are not written to the file.

For most icon images, More and Ed work about equally well with both FirstIconImage and FirstBitplanes. Pseudo-images too wide to fit on the screen can be displayed only with Ed, since More automatically wraps lines that are too long, destroying the image. With Ed, you can scroll to different parts of the pseudo- image with the cursor.

FirstHexBitplanes contains most of the icon data you need, such as the Gadget and Image widths and heights. It also has Bitplane0 and Bitplane1 printed in hex, as they appear in NewZAP. A printout of this file is sometimes useful when you want to see the exact position of the start and end of each bitplane in a load file displayed in NewZAP.

Below is the loop in Process2 that does the main job:

```
/* Convert bitplanes into hexadecimal and binary formats. */
/* Compute color register number for each image pixel. */
/* Make output files of bitplanes in binary & hex formats, */
/* and make pseudo-image, using color register numbers in */
/* place of pixels. */

e=''
f=''
g=''
h=''
i=''
DO UNTIL a=''
  PARSE VAR a c +1 a
  PARSE VAR b d +1 b
  h=h||C2X(c)          /* Hexadecimal */
  i=i||C2X(d)
  c=C2B(c)             /* Binary */
  d=C2B(d)
  e=e||c
  f=f||d
  x=c+d+d              /* Color reg. nums. */
  g=g||LEFT(zeros,8-LENGTH(x))||x
  END
```

'a' and 'b' contain Bitplane0 and Bitplane1. Each bitplane is nibbled away a byte at a time. First, each pair of bytes is translated to hexadecimal and added to 'h' and 'i'. Then, they are translated to binary, transforming each byte into a pattern of eight 0s and 1s, which are added to 'e' and 'f'. Next, the bit patterns are added together, as decimal numbers, in the line 'x=c+d+d'. 'd' is added twice, since each 1 bit in Bitplane1 is the equivalent of decimal 2. This decimal addition may produce less than eight digits, since any leading 0s will be dropped. The next line restores them, so that we have a sequence of eight decimal digits from 0 to 3 for each pair of bytes in the two bitplanes. The following code writes these data to the three files.

I was struck by how simple the ARexx code is, to do the rather complicated job I have just described. I started with two strings of characters, and wound up with strings of hex, binary, and decimal digits, all without having to do any data typing. The equivalent code in BASIC would be much less elegant.

Writing your ARexx Script

Once you have found exactly where in your load file each piece of data that needs editing is located, you have done the hardest part of the job. Writing an ARexx script to do the editing may seem somewhat daunting if you don't have much experience with the language, but it needn't be. You should not have to do that much work.

Print all of the NewIconXXX.rexx scripts that accompany this article, and look them over. Pick the one that comes closest to matching the job you need to do. If you need to replace one icon image in a load file similar to AmigaBASIC, in which you must edit the data parts of instructions that copy elements to structures, rather than the structures themselves, NewIconAmigaBASIC.rexx would be a good choice. If you need to replace one or more dual-image icons, NewIconsSpectraColorJr.rexx should be a good model. Make a copy of the script that comes closest to matching your needs, and edit it to suit your load file.

Once you have written a few scripts that work well, you have a chance to contribute to the Amiga community, by publishing them. Writing these scripts takes some technical knowledge, but anyone can use them. There are probably a lot of users out there who can't write their own scripts, but would love to use yours. There is no fortune to be made here, but you can gain a little fame, and the gratitude of at least a few. You will also have an opportunity to use the word 'copyright' and your name in the same sentence.

Old Business

The structure definitions in my previous article, Re Color, got messed up. Most of the lines were too long, and were wrapped to the beginning of the next line. The information is correct, but it is hard to read. Also, the previously published version of RecolorIcons.rexx uses a poor routine, which simply inverts all the bits in the bitplane, including the unused bits, to swap the colors of illegal single-bitplane icon images. Since these unused bits are ignored by the system when it draws an icon, this makes no difference from the user's point of view, but it violates the programming convention, which requires that all unused bits in each image line of a bitplane are to be cleared. I have included a revised version of RecolorIcons.rexx on this issue's disk. My revision evidently did not arrive in time to make it into the previous issue.

Wrapup

We have covered a lot of ground in these two articles, but we still have one more mile to go. The methods I have described so far will work on a lot of load files, but not on all of them. If you try to recolor or replace the icon images in SARGON_III, the chess program, you will find that these methods do not work. Not to worry. In my follow-up article, Re Color Plus, I will show you what does.

Listings

FindIconData.rexx

```
/****************** FindIconData.rexx *******************/
/*                                                      */
/*             Copyright 1993 by Dave Senger            */
/*                                                      */
/*                        Oct. 5/93                     */
/*                                                      */
/*   Please keep my name, this notice, and all comments, */
/* intact in any distribution of this program, in whole */
/* or in part.                                          */
/*                                                      */
/*  This ARexx script attempts to find Gadget and Image */
/* structures, and image Bitplanes, in load files.      */
/*                                                      */
/********************************************************/


/********************** U S A G E *********************/
/*                                                      */
/*    Assuming that ARexx is properly set up on your    */
/* system, and that this script is in your Rexxc        */
/* directory, CD a Shell to directory containing        */
/* your load file, then enter, for example:             */
/*                                                      */
/*       rx FindIconData MyProg Pathname/MyFile.info    */
/*                                                      */
/* Or just enter:                                       */
/*                                                      */
/*                    rx FindIconData                   */
/*                                                      */
/* and follow the prompts.                              */
/*                                                      */
/********************************************************/


PARSE ARG progname infofile
progname=STRIP(progname)
infofile=STRIP(infofile)

SAY
SAY "   This program attempts to locate Gadget and Image"
SAY "structures, and image bitplanes, in load files."

DO WHILE progname=''
   SAY
   SAY "   Enter pathname/file name of load file "||,
       "(<RETURN> to quit):"
   OPTIONS PROMPT " >>>-» "
   PARSE PULL progname
   IF progname='' THEN EXIT 0
   END

IF ~OPEN('infile',progname) THEN DO /* Try to open file */
   SAY
   SAY "Can't find "progname,
       ".  Sorry, no can do!"
   EXIT 20                         /* If no file, quit */
   END
x=STATEF(progname)                 /* Get file attributes */
PARSE VAR x type size .            /* Pick out file size */

IF size<100 THEN DO
   SAY
   SAY "Sorry, "progname" is too small."
   x=CLOSE('progname')
   EXIT 0
   END

DO WHILE infofile=''
   SAY
   SAY "   Enter pathname/file name of .info file "||,
       "(<RETURN> to quit):"
   OPTIONS PROMPT " >>>-» "
   PARSE PULL infofile
   IF infofile=‘’ THEN DO
      x=CLOSE(‘progname’)
      EXIT 0
      END
   END

IF UPPER(RIGHT(infofile,5))~=‘.INFO’ THEN DO
   SAY
   SAY “   You MUST provide an icon file with a .info suffix.  Sorry!”
   x=CLOSE(‘infile’)
   EXIT 0
   END

IF ~OPEN(‘iconfile’,infofile) THEN DO /* Try to open file */
   SAY
   SAY “Can’t find “infofile,
      ”.  Sorry, no can do!”
   x=CLOSE(‘infile’)
   EXIT 20                  /* If no file, quit */
   END

remainder=size
counter=-1
DO WHILE remainder>0
   counter=counter+1
   string.counter=READCH(‘infile’,MIN(65535,remainder))
   remainder=remainder-LENGTH(string.counter)
   END

x=CLOSE(‘infile’)

IF counter>0 THEN DO i=0 to counter-1
   j=i+1
   str.i=RIGHT(string.i,4000)||LEFT(string.j,4000)
   END

CALL GetIconData()

gadgetstring=”Possible gg_Width, gg_Height of Gadget “
gadgetstring=gadgetstring||”structure starting after:”
imagestring1=”Possible ig_Width, ig_Height of first Image “
imagestring1=imagestring1||”structure starting after:”
bitplanestring1=”Possible first image bitplane data starting after:”
imagestring2=”Possible ig_Width, ig_Height of second Image “
imagestring2=imagestring2||”structure starting after:”
bitplanestring2=”Possible second image bitplane data starting after:”

SAY
DO j=0 TO counter
   CALL FindString(string.j,gwhfat,gadgetstring,0)
   END

IF counter>0 THEN DO j=0 TO counter-1
   CALL FindString(str.j,gwhfat,gadgetstring,1)
   END

SAY
DO j=0 TO counter
   CALL FindString(string.j,iwhd1,imagestring1,0)
   END

IF counter>0 THEN DO j=0 TO counter-1
   CALL FindString(str.j,iwhd1,imagestring1,1)
   END

IF GADGHIMAGE THEN DO
   SAY
   DO j=0 TO counter
      CALL FindString(string.j,iwhd2,imagestring2,0)
      END

   IF counter>0 THEN DO j=0 TO counter-1
      CALL FindString(str.j,iwhd2,imagestring2,1)
      END
   END

SAY
DO j=0 TO counter
   CALL FindString(string.j,bitplanes1,bitplanestring1,0)
   END

IF counter>0 THEN DO j=0 TO counter-1
   CALL FindString(str.j,bitplanes1,bitplanestring1,1)
   END

IF GADGHIMAGE THEN DO
   SAY
   DO j=0 TO counter
      CALL FindString(string.j,bitplanes2,bitplanestring2,0)
      END

   IF counter>0 THEN DO j=0 TO counter-1
      CALL FindString(str.j,bitplanes2,bitplanestring2,1)
      END
   END
```

```
SAY
SAY “That’s all.  So long!”
EXIT 0

/***** Internal functions follow *****/

GetIconData:

   magic=READCH(‘iconfile’,2)      /* Start of DiskObject structure */
   IF magic~=X2C(E310) THEN DO      /* If not icon .info file, quit */
      SAY “`”curdirpath||separator||infofile||,
         “’ is not a true icon .info file.”
      x=CLOSE(‘iconfile’)
      RETURN
      END

   x=SEEK(‘iconfile’,10)           /* Gadget structure */
   gwhfat=READCH(‘iconfile’,10)
   flags=SUBSTR(gwhfat,5,2)     /* - in DiskObject structure */
   GADGHIMAGE=BITTST(flags,1)   /* Dual-image icon?? */

   WBDISK=1
   WBDRAWER=2
   WBGARBAGE=5
   x=SEEK(‘iconfile’,26)           /* DiskOjbect structure */
   type=C2D(READCH(‘iconfile’,1))  /* Does icon open window?? */
   IF type=WBDISK | type=WBDRAWER | type=WBGARBAGE THEN window=1
   ELSE window=0
   x=SEEK(‘iconfile’,17)           /* DrawerData structure exist?? */
   do_DrawerData=C2D(READCH(‘iconfile’,4))

   /* If icon opens a window when double-clicked on, */
   /* or even if it doesn’t open a window, but */
   /* contains a DrawerData structure, then SEEK */
   /* past DrawerData structure and into Image structure.  */
   /* Else, just SEEK into Image structure. */

   IF window | do_DrawerData~=0 THEN x=SEEK(‘iconfile’,68)
   ELSE x=SEEK(‘iconfile’,12)      /* SEEK to 3rd word of 1st icon - */
   CALL ReadImageStructure()        /* - Image structure, then read */
   IF depth<2 THEN SIGNAL IllegalIcon()
   CALL ReadBitplanes()

   iwhd1=iwhd
   bitplanes1=bitplanes

   IF GADGHIMAGE THEN DO        /* Do, if second image exists */
      IF depth>1 THEN x=SEEK(‘iconfile’,(depth-2)*bpLength+4)
      ELSE x=SEEK(‘iconfile’,4)   /* SEEK to 3rd word of 2nd - */
      CALL ReadImageStructure() /* - Image structure, then read */
      IF depth<2 THEN SIGNAL IllegalIcon()
      CALL ReadBitplanes()
      iwhd2=iwhd
      bitplanes2=bitplanes
      END

   x=CLOSE(‘iconfile’)             /* Close .info file  */
   RETURN

ReadImageStructure:

   iwhd=READCH(‘iconfile’,6)       /* 3rd word of icon Image structure */
   width=C2D(SUBSTR(iwhd,1,2))
   height=C2D(SUBSTR(iwhd,3,2))
   depth=C2D(SUBSTR(iwhd,5,2))
   wordWidth=(width+15)%16
   bpLength=wordWidth*height*2
   RETURN

ReadBitplanes:
                                /* SEEK past end of icon Image structure */
   startBitplane0=SEEK(‘iconfile’,10)
   bitplanes=READCH(‘iconfile’,bpLength*2)   /* Read bitplanes */
   RETURN

FindString:

   i=1
   DO WHILE i>0
      i=INDEX(ARG(1),ARG(2),i)
      IF i>0 THEN DO
         seekposition=i-1-ARG(4)*4000+(j+ARG(4))*65535
         SAY
         SAY ARG(3)
         SAY “byte “seekposition
         SAY “Hexadecimal “D2X(seekposition)
         i=i+1
         END
      END
   RETURN

IllegalIcon:

   SAY
   SAY “Illegal icon, with only one bitplane per image.  Try another.”
   x=CLOSE(‘iconfile’)
   EXIT 20
```

NewIconAmigaBASIC.rexx

```
/**************** NewIconAmigaBASIC.rexx *****************/
/*                                                       */
/*              Copyright 1993 by Dave Senger            */
/*                                                       */
/*                        Oct. 5/93                      */
/*                                                       */
/*   Please keep my name, this notice, and all comments, */
/* intact in any distribution of this program, in whole  */
/* or in part.                                           */
/*                                                       */
/*   This ARexx script attempts to copy a new icon image */
/* to AmigaBASIC load file.                              */
/*                                                       */
/*********************************************************/


/********************** U S A G E **********************/
/*                                                       */
/*    Assuming that ARexx is properly set up on your     */
/* system, and that this script is in your Rexxc         */
/* directory, CD a Shell to directory containing         */
/* your load file, then enter, for example:              */
/*                                                       */
/* rx NewIconAmigaBASIC AmigaBASIC Pathname/MyFile.info  */
/*                                                       */
/* Or just enter:                                        */
/*                                                       */
/*                  rx NewIconAmigaBASIC                 */
/*                                                       */
/* and follow the prompts.                               */
/*                                                       */
/*********************************************************/


/*   Changes program icon generated by AmigaBASIC */
/* load file, by replacing two image bitplanes with two */
/* bitplanes from an icon of your choice, and by editing */
/* width & height words in Gadget and Image structures, */
/* to match the equivalent values in the icon.  */

/*   Make sure you have at least one backup of your */
/* AmigaBASIC load file.  CD a Shell to the directory */
/* containing AmigaBASIC, and enter the command, "rx */
/* NewIconAmigaBASIC", without the quotes.  When */
/* prompted, enter complete pathname and file name of */
/* icon .info file you want to use.  You should get a */
/* message saying that the job is done.  Test patch by */
/* running AmigaBASIC and saving a dummy file.  */
/* Attached icon should look like icon you used.  */
/* There are 480 bytes of space for bitplanes in */
/* the AmigaBASIC load file, only 240 of which are */
/* used by the program's original icon, so you can */
/* copy a somewhat larger icon.  Script won't let you */
/* use an icon whose bitplanes are too big.  You can */
/* check your icon beforehand by running it through */
/* PrintIconBitplanes.rexx, then printing the output */
/* file, 'FirstHexBitplanes', in RAM:.  */

PARSE ARG progname infofile
progname=STRIP(progname)          /* Strip spaces from each end */
infofile=STRIP(infofile)

SAY
SAY "   This program attempts to edit gg_Width, gg_Height,"
SAY "and gg_Flags words, which are written to Gadget"
SAY "structure, and also ig_Width and ig_Height words,"
SAY "which are written to Image structure, and also"
SAY "replace image bitplanes, in AmigaBASIC load file."
SAY "Also, resets ig_ImageData pointer 240 bytes back."

DO WHILE progname=''
   SAY
   SAY "   Enter pathname/file name of AmigaBASIC "||,
       "load file (<RETURN> to quit):"
   OPTIONS PROMPT " >>>-» "
   PARSE PULL progname
   progname=STRIP(progname)       /* Strip spaces from each end */
   IF progname='' THEN EXIT 0
   END

IF ~OPEN('patchfile',progname) THEN DO /* Try to open file */
   SAY
   SAY "Can't find "progname,
       ".  Sorry, no can do!"
   EXIT 20                        /* If no file, quit */
   END

x=STATEF(progname)                /* Get file attributes */
PARSE VAR x type size .           /* Pick out file size */
```

```
IF size ~=103500 THEN DO
   SAY "Wrong file.  Sorry, no can do!"
   x=CLOSE('patchfile')
   EXIT 20
   END

DO WHILE infofile=''
   SAY
   SAY "  Enter full pathname and file name of .info file whose"
   SAY "bitplanes you want to copy to AmigaBASIC "||,
       "load file (<RETURN> to quit)."
   SAY
   OPTIONS PROMPT " >>>-» "
   PARSE PULL infofile
   infofile=STRIP(infofile)      /* Strip spaces from each end */
   IF infofile='' THEN DO
      x=CLOSE('patchfile')
      EXIT 0
      END
   END

IF UPPER(RIGHT(infofile,5))~='.INFO' THEN DO
   SAY
   SAY "File name must have a .info suffix.  Try again."
   x=CLOSE('patchfile')
   EXIT 20
   END

IF ~OPEN('infile',infofile) THEN DO /* Try to open specified file */
   SAY
   SAY "Can't find `"infofile"'.  Sorry, no can do!"
   x=CLOSE('patchfile')
   EXIT 20
   END

magic=READCH('infile',2)      /* Start of DiskObject structure */
IF magic~=X2C(E310) THEN DO   /* If not icon .info file, quit */
   SAY
   SAY "`"infofile"' is not a true icon .info file."
   x=CLOSE('patchfile')
   x=CLOSE('infile')
   EXIT 20
   END

x=SEEK('infile',10)           /* Gadget structure embedded - */
gg_width=C2D(READCH('infile',2))
gg_height=C2D(READCH('infile',2))
flags=READCH('infile',2)      /* - in DiskObject structure */
GADGHIMAGE=BITTST(flags,1)    /* Dual-image icon?? */
GADGBACKFILL=BITTST(flags,0) /* Backfill or complement?? */

WBDISK=1
WBDRAWER=2
WBGARBAGE=5
x=SEEK('infile',30)           /* DiskOjbect structure */
type=C2D(READCH('infile',1)) /* Does icon open window?? */
IF type=WBDISK | type=WBDRAWER | type=WBGARBAGE THEN window=1
ELSE window=0
x=SEEK('infile',17)           /* DrawerData structure exist?? */
do_DrawerData=C2D(READCH('infile',4))

/* If icon opens a window when double-clicked on, */
/* or even if it doesn't open a window, but */
/* contains a DrawerData structure, then SEEK */
/* past DrawerData structure and into Image structure.  */
/* Else, just SEEK into Image structure. */

IF window | do_DrawerData~=0 THEN x=SEEK('infile',68)
ELSE x=SEEK('infile',12)      /* SEEK to 3rd word of 1st - */
CALL ReadImageStructure()     /* - Image structure, then read */
IF bpLength>240 THEN DO
   SAY
   SAY "Bitplanes are each "bpLength" bytes long.  Max. is 240 bytes."
   SAY "Sorry, no can do."
   x=CLOSE('patchfile')
   x=CLOSE('infile')
   EXIT 20
   END
IF depth>1 THEN CALL Copy2()
ELSE DO
   SAY
   SAY "Only one bitplane.  Sorry, no can do."
   x=CLOSE('patchfile')
   x=CLOSE('infile')
   EXIT 20
   END

x=CLOSE('infile')             /* Close input .info file  */

/* First word written to load file below is written to second */
/* half of data longword of MOVE.L instruction.  Next 5 words */
/* are written to data words of MOVE.W instructions.  AmigaBASIC */
/* copies these words to Gadget and Image structures.  */
```

```
x=SEEK('patchfile',86260,'begin')  /* Set file pointer */
x=WRITECH('patchfile',D2C(498,2))  /* Reset ig_ImageData pointer */

x=SEEK('patchfile',86292,'begin')  /* Set file pointer */
x=WRITECH('patchfile',D2C(gg_width,2))  /* Gadget structure embedded - */
x=SEEK('patchfile',4)
x=WRITECH('patchfile',D2C(gg_height,2)) /* - in DiskObject structure */

x=SEEK('patchfile',4)
abGflags=READCH('patchfile',2)       /* Read AB Gadget flags word */
IF GADGHIMAGE THEN abGflags=BITSET(abGflags,0) /* Restore AB's default
comp. mode */
ELSE DO                              /* If icon being copied - */
   IF GADGBACKFILL THEN abGflags=BITSET(abGflags,0)
   ELSE abGflags=BITCLR(abGflags,0)  /* - isn't dual-image, give AB - */
   END                               /* - same complement mode as icon */

x=SEEK('patchfile',-2)               /* Set file pointer */
x=WRITECH('patchfile',abGflags)      /* Replace edited flags word */

x=SEEK('patchfile',86190,'begin')  /* Set file pointer */
x=WRITECH('patchfile',D2C(width,2))  /* Image structure */
x=SEEK('patchfile',4)
x=WRITECH('patchfile',D2C(height,2))

x=SEEK('patchfile',89266,'begin') /* Set file pointer */
x=WRITECH('patchfile',a)         /* Write bitplanes */
x=WRITECH('patchfile',b)

x=CLOSE('patchfile')             /* Close patched AB file - */
SAY "That gets it.  So long!"    /* - and get out */
EXIT 0

ReadImageStructure:

   width=C2D(READCH('infile',2)) /* 3rd word of icon Image structure */
   height=C2D(READCH('infile',2))
   depth=C2D(READCH('infile',2))
   wordWidth=(width+15)%16
   bpLength=wordWidth*height*2
   RETURN

Copy2:

   x=SEEK('infile',10)          /* SEEK past end of Image structure */
   a=READCH('infile',bpLength)  /* Read 2 bitplanes */
   b=READCH('infile',bpLength)
   RETURN
```

NewIconColors.rexx

```
/*****************  NewIconColors.rexx  ****************/
/*                                                     */
/*             Copyright 1993 by Dave Senger           */
/*                                                     */
/*                      Oct. 5/93                      */
/*                                                     */
/*   Please keep my name, this notice, and all comments, */
/* intact in any distribution of this program, in whole */
/* or in part.                                         */
/*                                                     */
/*   This ARexx script recolors icons in a variety of  */
/* ways.                                               */
/*                                                     */
/*******************************************************/


/********************** U S A G E *********************/
/*                                                     */
/*    Assuming that ARexx is properly set up on your   */
/* system, and that this script is in your Rexxc       */
/* directory, CD a Shell to a directory containing     */
/* icon(s) you want to recolor.  To recolor a single   */
/* icon, enter, for instance:                          */
/*                                                     */
/*           rx NewIconColors MyFile.info              */
/*                                                     */
/*    To recolor several icons, omit the file name, and */
/* follow the prompts.                                 */
/*                                                     */
/*******************************************************/

/*    Recolors .info files by swapping two bitplanes of */
/* image data, reversing black & white colors.  If .info */
/* file is for a dual-image type of icon, swaps second */
/* pair of bitplanes, as well.  Also offers several */
/* other choices.  */

/*   This version fixes problem with some icons which */
/* break the rules, such as the one that looks like */
/* a book with the title 'Doc File' on the cover, */
/* that appears in many Fred Fish disks.  This icon */
/* uses 3 bitplanes instead of two, so if the first Image */
/* structure specifies a depth of 3 or more, the script */
/* swaps only the first two bitplanes, and SEEKs past */
/* the additional bitplane(s) before reading the second */
/* Image structure if there is one, then swapping two */
/* more bitplanes. */

/*   The Third edition of the ROM Kernel Manual, LIBRARIES */
/* volume, specifies on page 353 that the image depth MUST */
/* be 2, and PlanePick MUST be 3.  This icon uses an image */
/* depth of 3, and sets PlanePick to 7 (all 3 least sig- */
/* nificant bits set, signifying that all 3 planes are to be */
/* used), breaking two of the rules for icons designed for */
/* pre-Version 3 'OS's.  Oddly enough, only the first two */
/* bitplanes for each image contain image data.  Each third */
/* plane is null.  */

/*   This version also works with icons which use */
/* single-bitplane images, which are also illegal.  */
/* Such images don't have two bitplanes to swap, so */
/* script inverts the single bitplane, flipping ones */
/* to zeros, and zeros to ones, which swaps the two */
/* colors.  */

/*   This version also works with icons, other than */
/* WBDISK, WBDRAWER, or WBGARBAGE, which have a */
/* DrawerData structure.  Script checks do_DrawerData */
/* longword for null, and, if not null, SEEKs past */
/* DrawerData structure to get into first Image structure.  */

line.1="swap black-white {1«-»2}                            "
line.2="swap grey-blue {0«-»3}                              "
line.3="swap black-white {1«-»2}, and also grey-blue {0«-»3}"
line.4="swap grey-black {0«-»1}                             "
line.5="swap white-blue {2«-»3}                             "
line.6="swap grey-black {0«-»1}, and also white-blue {2«-»3}"
line.7="swap grey-white {0«-»2}                             "
line.8="swap black-blue {1«-»3}                             "
line.9="swap grey-white {0«-»2}, and also black-blue {1«-»3}"
line.10="rotate colors forward {0-»1-»2-»3-»0}              "
line.11="rotate colors backward {0-»3-»2-»1-»0}             "

PARSE ARG infofile             /* Get file name, if any */

/* Check to see if user has specified a different */
/* directory.  If so, obtain pathname and file name, */
/* and change to new directory, if possible.  */

slash=LASTPOS('/',infofile)
colon=LASTPOS(':',infofile)
IF slash>0 THEN DO
   curdirpath=LEFT(infofile,slash)
   infofile=RIGHT(infofile,LENGTH(infofile)-slash)
   IF PRAGMA('d',curdirpath)='' THEN SIGNAL BadPath
   END
ELSE IF colon>0 THEN DO
   curdirpath=LEFT(infofile,colon)
   infofile=RIGHT(infofile,LENGTH(infofile)-colon)
   IF PRAGMA('d',curdirpath)='' THEN SIGNAL BadPath
   END

curdirpath=PRAGMA('d')  /* Get full pathname of current directory */
IF RIGHT(curdirpath,1)=':' THEN separator=''
ELSE separator='/'

IF infofile='' THEN only1icon=0
ELSE only1icon=1               /* If file name, set flag */

IF only1icon THEN DO
   IF UPPER(RIGHT(infofile,5))~='.INFO' THEN DO
      SAY
      SAY "File name must have a `.info' suffix.  "||,
         "Sorry, no can do!"
      EXIT 20                  /* Quit */
      END
   END
ELSE DO UNTIL choice=1 | choice=2
   SAY
   SAY "Your current directory is:"
   SAY
   SAY curdirpath
   SAY
   SAY "  You can recolor ALL ICONS IN THIS DIRECTORY ONLY (1),"
   SAY "or also ALL ICONS IN ALL CHILD DIRECTORIES (2)."
   SAY
   OPTIONS PROMPT "Enter 1, 2, or Q to quit.  >>>-» "
   PULL choice
   choice=LEFT(COMPRESS(choice),1)
   IF choice=''|choice='Q'|choice=X2C(1B) THEN EXIT 0
   END                      /* ESCape */  /* Quit */

IF ~only1icon THEN DO
   IF choice=2 THEN DO UNTIL YesNo='Y' | YesNo='N'
      SAY
      SAY "   Script will recolor ALL ICONS IN THIS DIRECTORY,"
      SAY "AND ALSO IN ALL CHILD DIRECTORIES."
      SAY
      OPTIONS PROMPT "   Is this what you want?  (Y/N) >>>—» "
      PULL YesNo
      YesNo=LEFT(COMPRESS(YesNo),1)
      END

   IF YesNo='N' THEN DO
      SAY
      SAY "No icons recolored."
      EXIT 0                    /* Quit */
      END
   END

pick=0
DO WHILE pick<1 | pick>11
   SAY
   SAY "      Hit <RETURN> to quit.  CHOOSE FROM..."
   SAY
   DO i=1 TO 11
      SAY UPPER(line.i)"  ("i")."
      END
   SAY
   OPTIONS PROMPT "Enter your choice.  >>>—» "
   PULL pick
   IF ~DATATYPE(pick,'w') THEN DO
      SAY
      SAY "You must enter an integer between 1 and 11."
      SAY
      SIGNAL GetOut
      END
   END

SAY
SAY "Script will "STRIP(line.pick)"."

SAY
IF only1icon THEN DO
   CALL SwapBitplanes()
   SAY
   SAY "That gets it.  So long!"
   EXIT 0                    /* Quit */
   END
ELSE CALL Recolor(curdirpath)

GetOut:

   SAY
   SAY "That gets it.  So long!"
   EXIT 0                     /* Quit */

/***** Internal functions follow *****/

Recolor: PROCEDURE EXPOSE choice pick only1icon

   PARSE ARG curdirpath        /* Get new directory pathname */
   IF RIGHT(curdirpath,1)=':' THEN separator=''
   ELSE separator='/'

   x=PRAGMA('d',curdirpath)     /* Change to new directory */

   files='/'||SHOWDIR('','f','/')||'/'   /* Get file list */
   ufiles=UPPER(files)          /* Make UPPER CASE copy */

   info=1
   DO WHILE info>0              /* Extract .info file names */
      info=INDEX(ufiles,'.INFO/',info)
      IF info>0 THEN DO
         slash=LASTPOS('/',ufiles,info)
         info=info+5
         infofile=SUBSTR(files,slash+1,info-slash-1)
         CALL SwapBitplanes()   /* Swap 2nd & 3rd colors, or swap — */
         END                    /* — 2 colors of only bitplane */
      END

   DROP files
   DROP ufiles

   IF choice=2 THEN DO
      dirs=SHOWDIR('','d','/')||'/'      /* Get directory list */
      DO WHILE dirs>'/'
         PARSE VAR dirs newdir '/' dirs  /* Build new pathname & — */
         nextdirpath=curdirpath||separator||newdir
         CALL Recolor(nextdirpath)       /* — Recolor new directory */
         END

      END
   RETURN

SwapBitplanes:

   Invert=1  /* 1 inverts single-bitplane images.  0 switches off */

   IF ~OPEN('patchfile',infofile) THEN DO /* Try to open specified file */
      SAY "Can't find `"curdirpath||separator||infofile||,
         "'.  Sorry, no can do!"
      IF only1icon THEN EXIT 20     /* Quit */
      RETURN
      END

   magic=READCH('patchfile',2)      /* Start of DiskObject structure */
   IF magic~=X2C(E310) THEN DO      /* If not icon .info file, quit */
      SAY "`"curdirpath||separator||infofile||,
         "' is not a true icon .info file."
      x=CLOSE('patchfile')
      IF only1icon THEN EXIT 0      /* Quit */
      RETURN
      END

   /* If you don't want .info files listed as they are processed, */
   /* disable next line.  */

   SAY "Recoloring "curdirpath||separator||infofile

   x=SEEK('patchfile',14)          /* Gadget structure embedded — */
   flags=READCH('patchfile',2)     /* — in DiskObject structure */
   GADGHIMAGE=BITTST(flags,1)      /* Dual-image icon?? */

   WBDISK=1
   WBDRAWER=2
   WBGARBAGE=5
   x=SEEK('patchfile',30)          /* DiskOjbect structure */
   type=C2D(READCH('patchfile',1))  /* Does icon open window?? */
   IF type=WBDISK | type=WBDRAWER | type=WBGARBAGE THEN window=1
   ELSE window=0
   x=SEEK('patchfile',17)          /* DrawerData structure exist?? */
   do_DrawerData=C2D(READCH('patchfile',4))

   /* If icon opens a window when double-clicked on, */
   /* or even if it doesn't open a window, but */
   /* contains a DrawerData structure, then SEEK */
   /* past DrawerData structure and into Image structure.  */
   /* Else, just SEEK into Image structure. */

   IF window | do_DrawerData~=0 THEN x=SEEK('patchfile',68)
   ELSE x=SEEK('patchfile',12)      /* SEEK to 3rd word of 1st — */
   CALL ReadImageStructure()        /* — Image structure, then read */
   IF depth>1 THEN CALL Swap2()
   ELSE IF Invert THEN DO
      CALL MakeMaskstr()
      CALL Invert1()
      END
   ELSE DO
      x=SEEK('patchfile',bpLength+10)
      SAY curdirpath||separator||infofile,
         "first single bitplane not inverted."
      END

   /* If icon uses 2 images, then swap second pair of */
   /* bitplanes (or invert second single bitplane), also. */

   IF GADGHIMAGE THEN DO            /* Do, if second image exists */
      IF depth>1 THEN x=SEEK('patchfile',(depth-2)*bpLength+4)
      ELSE x=SEEK('patchfile',4)    /* SEEK to 3rd word of 2nd — */
      CALL ReadImageStructure()     /* — Image structure, then read */
      IF depth>1 THEN CALL Swap2()
      ELSE IF Invert THEN DO
         CALL MakeMaskstr()
         CALL Invert1()
         END
      ELSE SAY curdirpath||separator||infofile,
            "second single bitplane not inverted."
      END

   x=CLOSE('patchfile')             /* Close patched file  */
   RETURN

ReadImageStructure:
   width=C2D(READCH('patchfile',2)) /* 3rd word of Image structure */
   height=C2D(READCH('patchfile',2))
   depth=C2D(READCH('patchfile',2))
   wordWidth=(width+15)%16
   bpLength=wordWidth*height*2
   RETURN

Swap2:
                              /* SEEK past end of Image structure */
   startBitplane0=SEEK('patchfile',10)
   a=READCH('patchfile',bpLength)   /* Read 2 bitplanes */
   b=READCH('patchfile',bpLength)

   INTERPRET 'CALL Routine'pick'()'

   x=SEEK('patchfile',startBitplane0,'begin')
   x=WRITECH('patchfile',a)         /* Write back 2 bitplanes */
   x=WRITECH('patchfile',b)
   RETURN


/********** F I R S T   R O U T I N E **********/
/* Reverses black & white colors (1 & 2), by swapping */
/* Bitplane0 & Bitplane1.  */

Routine1:

   c=a
   a=b
   b=c
   RETURN

/********** S E C O N D   R O U T I N E **********/
/* Reverses grey & blue colors (0 & 3).  */

Routine2:

   CALL MakeMaskstr()
   c=BITXOR(a,b)
   d=BITXOR(c,maskstr)
   a=BITXOR(a,d)
   b=BITXOR(b,d)
   RETURN

/********** T H I R D   R O U T I N E **********/
/* Reverses black & white colors (1 & 2), and also */
/* grey and blue colors (0 & 3).  */

Routine3:

   CALL MakeMaskstr()
   a=BITXOR(a,maskstr)
   b=BITXOR(b,maskstr)
   RETURN

/********** F O U R T H   R O U T I N E **********/
/* Reverses grey & black colors (0 & 1).  */

Routine4:

   CALL MakeMaskstr()
   c=BITXOR(b,maskstr)
   a=BITXOR(a,c)
   RETURN

/********** F I F T H   R O U T I N E **********/
/* Reverses white & blue colors (2 & 3).  */

Routine5:

   a=BITXOR(a,b)
   RETURN

/********** S I X T H   R O U T I N E **********/
/* Reverses grey & black colors (0 & 1), */
/* and also white & blue colors (2 & 3).  */

Routine6:

   CALL MakeMaskstr()
   a=BITXOR(a,maskstr)
   RETURN

/********** S E V E N T H   R O U T I N E **********/
/* Reverses grey & white colors (0 & 2).  */

Routine7:

   CALL MakeMaskstr()
   c=BITXOR(a,maskstr)
   b=BITXOR(b,c)
   RETURN

/********** E I G H T H   R O U T I N E **********/
/* Reverses black & blue colors (1 & 3).  */

Routine8:

   b=BITXOR(b,a)
   RETURN

/********** N I N T H   R O U T I N E **********/
/* Reverses grey & white colors (0 & 2), */
/* and also black & blue colors (1 & 3).  */

Routine9:

   CALL MakeMaskstr()
   b=BITXOR(b,maskstr)
   RETURN

/********** T E N T H   R O U T I N E **********/
/* Rotates grey to black, black to white, white to blue, */
/* & blue to grey (0 to 1, 1 to 2, 2 to 3, & 3 to 0).  */

Routine10:

   CALL MakeMaskstr()
   b=BITXOR(b,a)
   a=BITXOR(a,maskstr)
   RETURN

/********** E L E V E N T H   R O U T I N E **********/
/* Rotates grey to blue, black to grey, white to black, */
/* & blue to white (0 to 3, 1 to 0, 2 to 1, & 3 to 2).  */

Routine11:

   CALL MakeMaskstr()
   a=BITXOR(a,maskstr)
   b=BITXOR(b,a)
   RETURN

ReadImageStructure:

   width=C2D(READCH('patchfile',2)) /* 3rd word of Image structure */
   height=C2D(READCH('patchfile',2))
   depth=C2D(READCH('patchfile',2))
   wordWidth=(width+15)%16
   bpLength=wordWidth*height*2
   RETURN

MakeMaskstr:

   maskstr=COPIES('FFFF'x,wordWidth-1)||,
          D2C(65536-2**(wordWidth*16-width),2)
   maskstr=COPIES(maskstr,height)
   RETURN

Invert1:
                              /* SEEK past end of Image structure */
   startBitplane0=SEEK('patchfile',10)
   a=READCH('patchfile',bpLength)   /* Read only bitplane */
   a=BITXOR(a,maskstr)              /* Invert */
   x=SEEK('patchfile',startBitplane0,'begin')
   x=WRITECH('patchfile',a)         /* Write back */
   RETURN

BadPath:

   SAY
   SAY "Bad pathname.  Try again."
   EXIT 20                    /* Quit */
```

✔

Please Write to:
Dave Senger
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

A Simple AmigaDOS Handler

by Stephen Rondeau

One of the ways the power of AmigaDOS is revealed is through the way it handles input and output (I/O) of data. AmigaDOS passes high-level I/O requests to a "handler", which is a program that can process requests directed to it.

Such high-level handlers are also called "AmigaDOS devices", as opposed to Exec devices and logical devices. An Exec device is low-level software that directly operates on a real device; the floppy drive, for instance, is controlled by the trackdisk.device. Logical devices are those that the user defines using the <I>Assign<I> command.

A user typically wants some operations to be performed on a data object, and will provide a name for that object. The beginning part of that name, up to the first colon, is often the handler name. For example, "RAM:appointments" is a named data object whose handler is "RAM:".

Usually, a data object corresponds to a file, and the handler which processes requests for files is a file system handler. But there is no requirement that the data object be a file — it could simply be a series of bytes, such as those passed to the SER: or PAR: handlers, which manage the serial port and the parallel port, respectively. For those handlers, naming a data object is not necessary, though some aspect of opening and closing the Exec device must be performed before reading or writing.

Handlers and Packets

Handlers are primarily concerned with such core operations as opening and closing a named object and reading data from and writing data to a named object. However, there are many other types of requests that can be asked of the handler. One request can determine if a handler is a file system or not, since file systems have certain operations they must support, such as unique naming and returning information about existing files. Another request determines if a handler is interactive, such as a console window handler.

A request is embodied in an AmigaDOS packet. A description of all AmigaDOS packets can be found in the book <U>The AmigaDOS Manual, 3rd Edition<U>, published by Bantam Books. This book is the only source of detailed information on packets.

The <I>DosPacket<I> structure describes the components of the packet, and can be found in the AmigaDOS include files. The key fields in that structure are:

* the type of packet, or "action" to be taken
* the argument fields, whose contents depend on the type of packet

Handler Operation

A handler basically consists of an event-processing loop, where the events are the arrivals of packets on the handler's message port. Exec messages contain the address of a packet in the message node's name field. After a message has arrived, the packet address is extracted and the type of the packet selects the code fragment that will process this request. When processing is complete, the packet with return value information is essentially returned to the requesting program, and the handler waits for the arrival of the next packet.

A handler is selective about what types of packets it processes. When it doesn't know how to process a packet, it returns that information to the requesting program. For packets that it can process, the results returned typically reflect the success of the operation, but some packets will return a data value. For example, a write request will return the number of bytes written; a read request will return the number of bytes read as well as directly fill the buffer whose address was found in one of the packet's arguments.

File Name Generator Design

To illustrate a very simple handler, the problem of generating an output file name will be used. Consider the situation where you are creating a series of pictures for an animation, and your paint package does not support sequential numbering of each picture. Or perhaps you want the date and time in the name of all of the files you create, so that you are assured of the time of their creation. Embedded date-time information can also allow the use of familiar file pattern-matching operations as well as easy sorting of files by date of creation. Sometimes it is useful to be able to substitute a "static" name from an environment variable.

Given those requirements of embedding a static name, sequence number, or date-time information, a way to specify this is required. A technique similar to the C library's printf() format string will be used. This involves a special marker character which may indicate the position of where one of those items is to be substituted. It also indicates that the character following it can be a code for the type of item to substitute.

Let's use the percent character "%" as the marker, and the following characters as item codes:

n — static name

s — sequence number

d — date-time

For example, in a name specification, the user would type "%s" wherever the sequence number was desired.

The actual name specification is somewhat more complicated than that. First, it must include the handler name and its trailing colon. Let's call our handler "ng", for name generator. One could write a name specification such as "ng:pic.%s", and expect to generate files named, for example:

```
SYS:pic.1
SYS:pic.2
SYS:pic.3
```

Why is "SYS:" used and not the current directory? A handler can be called from a program run from either a CLI or the Workbench. Only CLIs have current directories, so the default directory is always "SYS:" for handlers. Therefore, if the user wants a specific existing file directory, the full path to the file must be used. If the path is "Work:artwork", then a name specification given as "ng:Work:artwork/%n.%d" may result in file names such as:

Work:artwork/joes_grill.19940405@104523 Work:artwork/joes_grill.19940418@153254

Source of Substituted Items

But from where do these items to substitute come? For static names and sequence numbers, they could be saved in files, but a simpler way is to use environment variables. Environment variables can be read and written by ANSI C library functions in the handler. A value is read by getenv() and written by putenv(). Alternatively, the dos.library functions GetVar() and PutVar() can be used.

For our purposes, "NG_NAME" will be the name for the variable containing the static name value, which would be "joes_grill" in the preceding example.

"NG_SEQ" is the name of the sequence number environment variable. If it does not exist, the sequence starts at zero. If it does exist, the existing number is incremented by one. In either case, the resulting sequence number is assigned to "NG_SEQ".

Date and time information can be supplied either by the ANSI C library function strftime(), SAS/C's getclk() function, or, with some effort, the timer.device's GetSysTime() function combined with the utility.library's Amiga2Date() function.

Handler Magic

The problem of name generation will make the job of constructing a handler easy, since we will only be dealing with three types of packets.

The dos.library's Open() function is invoked from the requesting program, and starts the packet-sending process. It extracts the handler name and sends an ACTION_STARTUP packet to it. That packet contains a pointer to the portion of the name specification that follows the handler name's colon. Our handler is supposed to do initialization at this point, but what is desired for later processing is any destination handler name that followed ours.

For example, if "&ng:Work:artwork/%n.%d" had been specified, the requesting program's Open() would have stripped off the "ng:", and passed "Work:artwork/%n.%d" with the packet. Our handler would then extract and save "Work:" so that we can use it later on.

The order and type of subsequent packets depends on the requesting program. However, most programs try to determine if the handler is a file system, using the dos.library IsFileSystem() function. The ACTION_IS_FILESYSTEM packet does not contain any information on the handler being checked, since it is sent to our handler and it should know if it is a file system or not. In our peculiar situation, we want to check if our generated file name belongs to a file system. This is why in ACTION_STARTUP we parsed the destination handler's name. As a result, IsFileSystem() is called from our handler with the destination handler's name, and the result of that function is returned to the requesting program.

When a file is opened for output, the name of the file plus the MODE_NEWFILE constant is passed to the Open() function. That is what the requesting program will eventually send, as a packet type of ACTION_FINDOUTPUT. The name specification following our handler name's colon is passed in the packet. Now we search that string for the markers and perform the substitutions, generating the new file name.

The new file name is used by our handler in an Open() function call. The intent is to get access to the destination handler for that new file.

In general, the Open() function allocates and returns a file handle which includes a handler's address. The requesting program's file handle, whose address is in an ACTION_FINDOUTPUT packet argument, is filled in using the new file's handle information. Then the storage for the new file handle's information is freed, as the requesting program's file handle will be used from now on.

That's where all of the magic exists in this technique of intercepting a name specification and generating a new name from it. Since file operations always consult the handler address within the file handle, any subsequent operations will be directed to that handler — our handler won't see them.

Any packets other than those described will be considered unknown packets.

While that's all this simple handler will process, if one wants a complete name generator handler, other packets must be processed. Some independent operations such as setting the protection bits and deleting a file are sent as separate requests, directly to our handler. This occurs because the requesting program only knows about our handler name and the name specification; it does not know about the generated file name. These requests, as well as any other not described in this article, will be flagged as unknown actions by this simple handler. In a more complete handler, those requests should be handled as well. Saving the original name specification, a unique identifier for the requesting program, and the generated file name would make it relatively easy to handle these independent requests.

Compilation and Installation Instructions

SAS/C v6.0 was used to develop the supplied source code. To compile it, change the directory to the "ng" source directory, which is called "Work:ng/source" here, and type:

```
    smake
```

It should create a file called "ng-handler" in that directory.

To install it, you have to add the source directory to the "devs:" list and mount the handler. To add the directory:

```
   Assign DEVS: Work:ng/source ADD
```

That command should be placed in your "S:user-startup" file.

The "mountlist.ng" file can be found in that source directory and looks like this:

```
    NG:       Stacksize  = 8000
              Priority   = 5
              GlobVec    = -1
              Handler = devs:ng-handler
    #
```

You shouldn't have to change it if the ASSIGN statement above is executed before you mount the handler. Mounting the handler should also appear in your "S:user-startup" file:

```
    mount ng: from Work:ng/source/mountlist.ng
```

Once mounted, the handler is ready to use. However, you may want to initialize the environment variables NG_NAME and NG_SEQ before you use the handler. For example:

```
    setenv NG_NAME bio
    setenv NG_SEQ 1
    copy env:gn_#? envarc:
```

This sets the values and preserves them so they are available after turning on your Amiga.

Conclusion

AmigaDOS handlers are much more powerful than illustrated here with the simple name generator handler. They are also somewhat tricky to program. In more complicated handlers, one mimics the information returned from file systems, filling in such structures as file information blocks and file locks. The simple technique presented above, that of opening a file using the Open() function from within a handler, does not generalize easily for other dos.library functions.

Yet, once one masters the technique of writing robust handlers, they can be used to extend the functionality of any program that does I/O. This general applicability is one of the reasons why AmigaDOS is a powerful operating system.

Listing

```
/* ng-handler.c

   Copyright 1994 Stephen B. Rondeau.
   All Rights Reserved.

   File name generator handler:
     creates file names from a given format.

   Note that if you define the DEBUG value, you must
   have access to debug.lib during linking (available
   from Commodore) and either the serial port properly
   connected or sushi started during execution to see
   kprintf()'s output.
*/

#define __USESYSBASE

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <string.h>
#include <ctype.h>

#include <exec/types.h>
#include <exec/nodes.h>
#include <dos/dos.h>
#include <dos/dosextens.h>
#include <proto/dos.h>
#include <proto/exec.h>

#define MAX_PATH_SIZE 1024

LONG _ProgramName=NULL;
LONG _OSERR=0;

struct ExecBase *SysBase;
struct DosLibrary *DOSBase = NULL;

void Cleanup(void);
void GenFileName(struct DosPacket *, char *);

__saveds void
main(void)
{
  BOOL quit=FALSE;
  struct DosPacket * packet;
  struct Process *this_task;
  struct MsgPort *my_port;
  struct Message * message;
  LONG result1, result2;
  BPTR file;
  struct FileHandle * this_file;
#if DEBUG
  char debug_buffer[512];
#endif
  char name[MAX_PATH_SIZE+120];
  char callers_devname[100];
  char * colon_pos;
  ULONG signals, signals_to_check;

  SysBase = *((struct ExecBase **) 4);

  /* Use AmigaDOS 2.04 or higher since ReplyPkt() and
   * other 2.04-specific functions will be used.
   */
  DOSBase = (struct DosLibrary *)
            OpenLibrary(DOSNAME, 37L);
  if (DOSBase == NULL)
    goto cleanup;

  /* Get this process's address so that we can wait
   * on our port for a message/packet from someone.
   */
  this_task = (struct Process *) FindTask(NULL);
  my_port = &(this_task->pr_MsgPort);

  signals_to_check = (1L << my_port->mp_SigBit);

  /* The event-handling loop.
   *
   * There is only one event: a packet arriving at
   * our port. If it is a packet type of interest,
   * it is processed and valid values are returned.
   * Otherwise, the requesting process is told that
   * this handler does not understand the packet.
   */
  while (quit == FALSE)
  {
    signals = Wait(signals_to_check);

    WaitPort(my_port);
    message = GetMsg(my_port);
    if (message == NULL) /* If signal without message */
      continue;

    packet = (struct DosPacket *) message->mn_Node.ln_Name;

    switch (packet->dp_Type)
    {
      case ACTION_NIL:
#if DEBUG
strcpy(debug_buffer,"ACTION_NIL: ");
strncpy(debug_buffer+strlen(debug_buffer),
        BADDR((char *)packet->dp_Arg1+1),
        *(unsigned char *) BADDR(packet->dp_Arg1));

kprintf("%s\n",debug_buffer);
#endif
        /* Extract the device name, if any. */

        strncpy(callers_devname,
                BADDR((char *)packet->dp_Arg1+1),
                *(unsigned char *) BADDR(packet->dp_Arg1));
        if ((colon_pos = strchr(callers_devname, ':')) ==
             NULL)
          callers_devname[0] = '\0';
        else
        {
          colon_pos++;
          *colon_pos = '\0';
        }

        result1 = DOSTRUE;
        result2 = 0;
        break;

      case ACTION_IS_FILESYSTEM:
#if DEBUG
kprintf("ACTION_IS_FILESYSTEM\n");
#endif
        /* Use the extracted device name to check. */

        result1 = IsFileSystem(callers_devname);

#if DEBUG
kprintf("ACTION_IS_FILESYSTEM %s result=%ld\n",
        callers_devname, result1);
#endif
        result2 = 0;
        break;

      case ACTION_FINDOUTPUT:
#if DEBUG
kprintf("ACTION_FINDOUTPUT: name=%s\n",
        BADDR((char *)packet->dp_Arg3+1));
#endif
        /* Actually generate the file name from the spec
         * given.
         */
        GenFileName(packet, name);

        if (name[0] == '\0') /* If empty, error occurred */
        {
          result1 = DOSFALSE;
          result2 = ERROR_INVALID_COMPONENT_NAME;
        }
        else
        {
#if DEBUG
kprintf("name=%s\n", name);
#endif
          file = (long) Open(name, MODE_NEWFILE);

          if (file == NULL) /* If Open failed */
          {
            result1 = DOSFALSE;
            result2 = IoErr();
          }
          else
          {
            this_file = (struct FileHandle *) BADDR(file);

            /* Copy the Open() file handle info into
             * the requester's handle.
             */

            *((struct FileHandle *)
              BADDR(packet->dp_Arg1)) = *this_file;

            /* Free the original file handle Open() created.
             */

            FreeMem((char *) this_file,
                    sizeof(struct FileHandle));

            /* After the packet is returned, the program
             * will do I/O directly to the proper handler,
             * not this one.  However, additional
             * operations to that file, such as setting
             * the file protection bits or deleting the
             * file, would be routed to this handler.
             * These requests are simply ignored by this
             * handler.
             */
            result1 = DOSTRUE;
            result2 = 0;
          }
        }
        break;

      default:
#if DEBUG
sprintf(debug_buffer, "Unknown Packet type=%d\n",
        packet->dp_Type);
kprintf("%s",debug_buffer);
#endif

        result1 = DOSFALSE;
        result2 = ERROR_ACTION_NOT_KNOWN;
        break;
    }

    /* Return packet to originator. */

    ReplyPkt(packet, result1, result2);

  } /* Loop to catch packets */

cleanup:
  Cleanup();
}

void
Cleanup(void)
{
  if (DOSBase != NULL)
    CloseLibrary((struct Library *) DOSBase);
}
```

```
/* From the name specification pointed to by the packet's
 * third argument field, find the special markers and
 * codes and substitute the values associated with them
 * into the generated file name.
 */

void
GenFileName(struct DosPacket * packet, char * name)
{
  char marker='%';
  ULONG sequence_num; /* unsigned allows max sequencing */
  char number_spec[22]; /* Space for NG_SEQ=integer */

  char * env_value;    /* will point to env. var. value */

  char name_spec[256]; /* limited by BSTR length field */
  char * last_pos;
  char * pos;

  char * output_pos = name; /* pointer within gen. name */

  unsigned char clock[8];  /* for getclk() results */
  char dateinfo[20];       /* To hold yyyymmdd@hhmmss */

  BOOL have_seq=FALSE,     /* allows reuse of seq. num. */
       have_datetime=FALSE; /* reuse of date-time info */

  /* Copy the original name specification so that a
   * NUL terminator can be added.
   */

  strncpy(name_spec, BADDR((char *) packet->dp_Arg3+1),
          *(unsigned char *) BADDR(packet->dp_Arg3));

  name_spec[*(unsigned char *)BADDR(packet->dp_Arg3)]='\0';

  last_pos = name_spec;
  pos = last_pos;

  /* Look through all of the characters of the name
   * specification for the marker.  Then substitute the
   * associated value if the one-character code following
   * the marker is recognized.
   */
  pos = strchr(pos, marker);

  while (pos != NULL)
  {
    if (pos != name_spec) /* If marker not first char */
    {
      strncpy(output_pos, last_pos, pos-last_pos);
      output_pos += pos-last_pos;
    }

    if (pos+1 < name_spec+strlen(name_spec))
    {
      switch(toupper(pos[1]))
      {
      case 'N':  /* Insert static name; defaults to "ng" */
        if ((env_value = getenv("NG_NAME")) == NULL)
          env_value = "ng";

        strcpy(output_pos, env_value);
        output_pos += strlen(env_value);
        break;

      case 'S':  /* Insert the next sequence number */
        if (have_seq == FALSE)
        {
          have_seq = TRUE;

          if ((env_value = getenv("NG_SEQ")) == NULL)
            sequence_num = 0;
          else
          {
            sequence_num = strtoul(env_value, NULL, 10);
            sequence_num++;
          }

          /* Update the env. var. sequence number */

          sprintf(number_spec, "NG_SEQ=%lu", sequence_num);

          if (putenv(number_spec) != 0)
          {
            name[0] = '\0';
            return;
          }
        }

        strcpy(output_pos, number_spec+7);
        output_pos += strlen(number_spec+7);
        break;

      case 'D':  /* Insert date and time information */
        if (have_datetime == FALSE)
        {
          have_datetime = TRUE;

          getclk(clock);

          sprintf(dateinfo, "%d%02d%02d@%02d%02d%02d",
                  1980+clock[1],         /* year */
                  (short int) clock[2],  /* month */
                  (short int) clock[3],  /* day */
                  (short int) clock[4],  /* hour */
                  (short int) clock[5],  /* minute */
                  (short int) clock[6]); /* second */

        }
        strcpy(output_pos, dateinfo);
        output_pos += strlen(dateinfo);
        break;

      default:
        strncpy(output_pos, pos, 2);
        output_pos += 2;
        break;
      } /* End switch on type of value desired */

      last_pos = pos+2;
      pos += 2;        /* Skip over marker and code */
    }
    else /* marker is last character */
    {
      *output_pos = *pos;
      output_pos++;
      last_pos = pos+1;
      break;
    }

    pos = strchr(pos, marker); /* find next marker */
  }

  /* Copy any remaining characters to generated name. */

  if (last_pos < name_spec+strlen(name_spec))
  {
    strncpy(output_pos, last_pos,
            name_spec+strlen(name_spec)-last_pos);
    output_pos += name_spec+strlen(name_spec)-last_pos;
  }

  name[output_pos-name] = '\0'; /* Add terminating NUL */
}
```

✓

Please Write to:
by Stephen Rondeau
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

A PAIR OF PICKOVERS

by Bill Nee

One of my favorite authors of computer books is Clifford A. Pickover. He includes numerous topics in each book along with diagrams, pictures, and some programs. A lot of his programs, however, are pseudo-code, designed to give you an idea of how the program works. From his many books ("The World of Chaos", "Computers, Patterns, Chaos and Beauty", "Computers and the Imagination", etc.) I've selected two articles and adapted their programs for the Amiga. I'll give you a Basic listing for each program since that's easier to understand and then discuss the first assembly listing program in detail. You'll need the MATHIEEEDOUBBAS.LIBRARY and an Amiga 3000 (or one adapted to use a floating-point math co-processor).

QUATERNIONS

In previous articles I've graphed the Mandelbrot/Julia Sets using the complex values X+iY where "i" is the square root of -1. Now we'll use 4-dimensional complex numbers and graph them, but only in 2-dimension. These "hyper" numbers are called Quaternions (Q). First used by William Hamilton in the early 1800's, the numbers are current enough to be used in the space shuttle program. Since a Q number represents a 4-dimensional space, it must have four parts - a real value and three imaginary ones; the three imaginary ones are "i", "j", and "k". So a quaternion can be expressed as:

Q=A0+A1*i+A2*j+A3*k

When multiplying quaternions you must use these special rules:

i*i=j*j=k*k=i*j*k=-1 ij=-j*i=k j*k=-k*j=i k*i=-i*k=j

This means that when you square a Q it's new real portion is A0*A0-A1*A1-A2*A2-A3*A3 and it's new imaginary portion is 2*A0*A1i+2*A0*A2j+2*A0*A3k.

Now, this is very similar to squaring X+Yi as in the Mandelbrot/ Julia Sets and, in fact, the program we'll discuss will display the continued squaring of a Q value with another constant q added each time, just like the Julia Set. Also, the entire display is within a -2 to +2 in all directions. So our program can be summarized as Q=Q*Q+q. Unfortunately, it takes about 13 typewritten pages to list the program for this one-line equation. First, let's look at a Basic program that will graph a Q set.

You need to determine the left and right boundaries in the X direction and the top and bottom boundaries in the Y direction. The scale for each direction is the difference in boundaries divided by the display size (320X400). Each point within your boundaries is squared and a constant q added. If the absolute relative value of Q (A0*A0 + A1*A1 + A2*A2 + A3*A3) is greater than 4, then the point is outside the Q set and is colored according to the iteration count. If the point is still within the Q set after the maximum iteration count of 128, it is skipped (or you can give it a special Q set color).

Q IN BASIC

Using some of the values given by Mr. Pickover, A0 is the current X value, A1 is .05, A2 is the current Y value, and A3 is .05; the q values added are q0=-.745, q1=.113, q2=.01, and q3=.01. In Basic this program would be written as -

```
READ XLEFT,XRIGHT:XSCALE=(XRIGHT-XLEFT)/320
READ YBOTTOM,YTOP:YSCALE=(YTOP-YBOTTOM)/400
READ Q0,Q1,Q2,Q3
FOR H=0 TO 319:X=XLEFT+H*XSCALE
FOR V=0 TO 399:Y=YBOTTOM+V*YSCALE
A0=X:A1=.05:A2=Y:A3=.05
FOR C=1 TO 127
A0SQR=A0*A0:A1SQR=A1*A1:A2SQR=A2*A2:A3SQR=A3*A3
IF A0SQR+A1SQR+A2SQR+A3SQR>4 THEN COLORIT
A1=2*A0*A1+Q1:A2=2*A0*A2+Q2:A3=2*A0*A3+Q3:'new
imaginary part
A0=A0SQR-A1SQR-A2SQR-A3SQR+Q0:'new real part
NEXT C
LOOP1:NEXT V,H
STOP
COLORIT:CO=C\4
PSET(H,V),CO:GOTO LOOP1
DATA -1.5,1.5,-.8,.8,-.745,.113,.01,.01
```

Notice that I computed the imaginary portion first since it uses A0 and then computed the real portion. At this point you might want to try this program using Basic, HiSoft Basic, etc. Reduce the picture size (128X128) for a quicker look.

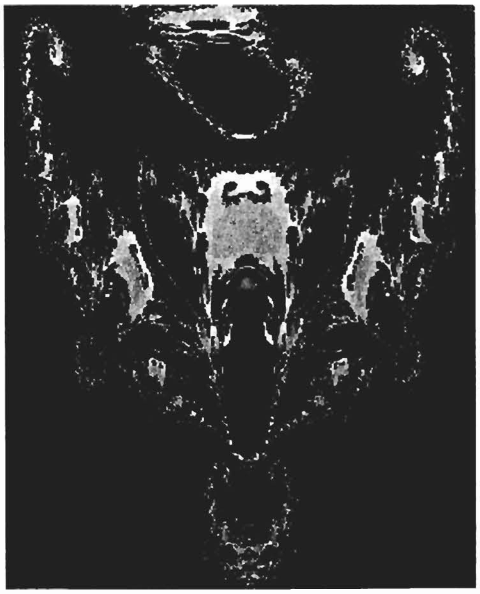


Figure 1
The Quaternion picture.

Now let's see how to accomplish this using an Assembly Language program. In this program you can input the beginning and ending coordinates (left, right, top, and bottom), select an A1/ A3 value, and pick the q to be added. While the program is drawing you can choose any of ten palettes using the Function keys, color cycle up or down with the corresponding arrows, create a zoom box using the LMB and dragging it, or quit the program. With ten palettes and 31 colors in each that's 310 different versions of one display! And you can cycle palettes even when the picture is completed to find the one you want to save. I've enclosed two pictures made from the listings in this article on the magazine disk*.

I'll be using the assembler PHXASS and the linker PNXLNK, both PD programs by Frank Wille. I like these programs so much that I registered with Mr. Wille and have received several updates. I did have to change my MENU.I file a little since PHXASS will presently only accept 10 parameters in a macro; the modified version is called PMENU.I. This latest assembler version corrects most of the items I mentioned in a previous article - quotes are no longer needed around include files, labels can start with "@", etc.

LISTING1

Take a look at Listing1. The first two lines are pseudo-code for the assembler. PHXASS expects the program to be written in sections with at least a minimum of a code section. FPU lets the assembler know we'll be using the floating-point registers and to accept those commands. Next, the six include files are listed. The variables COUNT, ACROSS, and DOWN are equated to the address registers a2, a3, and a4; I used address registers since the data registers are pretty busy with messages, menus, and zoom routines.

Next there are three macros. @PALETTE will load any color table (0-9) as a palette while @NEWPALETTE not only loads a color table but saves all 32 colors so they can easily be cycled. And @PSET stores the ACROSS and DOWN values in d0 and d1 and sets them. The DOWN value is subtracted from 399 to make it numerically correct. This wouldn't be necessary if there wasn't going to be a zoom routine.

After saving the STACK POINTER location the programs opens the INTUITION, GRAPHICS, DOS, and MATHIEEEDOUBBAS Libraries. The floating-point register commands themselves do not call for the math library (although they use it), but my routine to convert an ASCII string to a double-precision number does require it.

The OPTIONS routine uses DOS commands to clear the screen, move the cursor, and print some string messages. Notice that it is not necessary to compute the string length since the macro does that for you. The various messages are a reminder of what quaternions are, the current default values, and what input you have. Press the LMB to continue the program. You can skip this portion by removing the ";" above OPTIONS and jumping directly to SETUP.

In SETUP the program opens the screen and window, sets the mode to JAM2, uses COLORTABLE2 as the current palette, and activates the menu. Since the list of gadgets is included in the window flags, they will automatically appear when the window opens. Select the coordinates you want to use along with the four q's. Then MSG_CHECK will determine if you've selected a menu (there is only menu0). If so, EVAL_MENUNUMBER will determine which item you've picked, if any. ITEM2 will quit the program; ITEM1 is not active at this time, and ITEM0 will evaluate the coordinates and start drawing. Any other choices or actions will return to MSG_CHECK.

The first step in COORDINATES is to remove the gadgets from the current window by substituting a zero in the window's gadget location. Then the ASCII strings in each gadget's buffer are converted to double-precision numbers and stored in their proper locations. Notice the extensive use of MOVEDP to shift the double long-word values. I added the gadget MIN COUNT; it's value is a whole number stored in b. Any point outside the Q set will not be plotted if the iteration count is less than b; this will reduce a lot of block coloring around the more interesting portions of the display. The initial b value for the current coordinates is 12.

Next, the coordinates are used to compute an XSCALE and YSCALE. Then the program branches to SHOWIT skipping over the CONVERT routine used to make numbers out of the gadget string buffers. CONVERT could have been a macro but then this long routine would have been inserted into the program ten times. Since speed isn't that critical at this point I made it a sub-routine.

FLOATING-POINT REGISTERS

The first step in SHOWIT is to put a #1 in the ALLDONE flag. Other routines can test this flag to see if they should continue drawing or not. The RastPort is stored in a1 and the GfxBase in a6. After setting DOWN and ACROSS to 0, the current X location is stored in fp7, the current Y location in fp5, and the constants in fp6 and fp4. Throughout the program the final floating-point register values will be -

FP7 = A0	FP3 = A0SQUARE
FP6 = A1	FP2 = A1SQUARE
FP5 = A2	FP1 = A2SQUARE
FP4 = A3	FP0 = A3SQUARE

The squares of fp7 through fp4 are then stored in fp3 through fp0.

Fp0 is saved in VARIABLE since we're going to temporarily change it's value. The sum of the squares is stored in fp0, moved to d0, compared to 4 and the original value of fp0 is restored from VARIABLE. If d0 is less than 4 the program continues, but if it's equal to or greater than 4 the point must be outside of the Q set. The current iteration count is compared to the minimum count value in b. If it's lower, the program branches to the next point. If not, that point is PSET using a color value of the count divided by 4; the program then goes on to the next point.

If the sum of the coordinates squared is less than 4, however, new coordinates must be computed. A0 in fp7 is added to itself since 2*a0 is used to compute each imaginary coefficient. This value is in turn multiplied by the values in fp6 through fp4 (A1 through A3); also, the q value for each coefficient (q1 through q3) is added. Since these q values aren't in a fp register, they must be added as ".d" (double-precision) constants. To compute the new real portion, A0, the squares of A1 through A3 are each subtracted from A0SQUARE and q0 is added. This value is moved back into fp7 as the new A0.

The count is increased by 1 and if it's less than 128 the whole procedure repeats, checking and computing new coordinates. When the count reaches 128 the point is assumed to be in the Q set and can be ignored, leaving it the background color, or it can be gives a special color and PSET; my program leaves it as the background color. Then the current X increment is added to the current X location.

MESSAGE CHECKS

At this point, there needs to be a message check to see what you want to do, if anything. If you select a menu, CHECK_MENU and EVAL_MENUNUMBER will see which menu and item you selected. If you choose item1 the program will clear the screen and window and branch to the start of the program where the window was initially opened. Or you can quit the program by selecting item2.

If you press any of the Function keys (rawkey codes $50 - $59) the program will use the corresponding color table as the new palette. The up or down arrow keys ($4C, $4D) will color-cycle up or down leaving the background color the same. Colors 1 through 31 are all moved up or down one space in COLORTABLE. Finally, by pressing the LMB down and dragging it across the screen, you can create a zoom box. Once you release the button the zoom routine will automatically begin. After computing new starting and ending coordinate, they're stored in their proper locations. The program also increases the minimum PSET count value in b by 5 since a zoom enlarges the detail. Then the program branches back to SCALE and begins drawing your enlarged display.

If you don't do anything the program checks to see if it's all the way across the row and, if so, increases the YBOTTOM value by YINC and draws the next row. When all 399 rows have been drawn, the ALLDONE flag is cleared and the program waits for you to select some option. Again, you can pick new values, change palettes, color-cycle, zoom, or quit. If you cycle the colors, that routine checks ALLDONE to see if you've completed the drawing. If you quit, the screen, window, and libraries are closed and the Stack Pointer restored.

At the end of this program my window flags are: mousebuttons!menupick!mousemove!rawkey and activate!smartrefresh!borderless!mousereport. The gadget macros have been reduced to only allow for the name, next gadget, X location, Y location, and identifying number. The gadget buffer contains the starting values that will draw a complete display. This program is on the magazine disk* as QUATERF.ASM and QUATERF. If you make any changes, reassemble it using PHXASS QUATERF.ASM and PHXLNK QUATERF.O.

QUARTIC VARIATIONS

The next program discusses Quartic Variations. This is a 2-dimensional representation of a 3-dimensional complex plot. There are three complex variables - X, Y, and Z each with a real and imaginary part. Initially the starting coordinates make up Z; a new Z is computed and used to make a new X while the old Z is used to make a new Y. A constant Q is always added to Z and X. The absolute value of Z is compared to 4 and if it's greater, that point is colored according to it's iteration count. After reading the coordinates and Q values the Basic program for this listing looks like -

```
FOR H=0 TO 319:X=XLEFT+H*XSCALE
FOR V=0 TO 399:Y=YBOTTOM+V*YSCALE
ZR=X:ZI=Y:XR=0:XI=0:YR=0:YI=0
FOR C=1 TO 127
ZZR=ZR*ZR:ZZI=ZI*ZI:IF ZZR+ZZI>4 THEN COLORIT
ZREAL=ZZR-ZZI-.5*ZR+QR
ZIMAG=2*ZR*ZI-.5*ZI+QI
ZR=ZREAL:ZI=ZIMAG
XR=ZR*ZR-ZI*ZI-.5*YR+QR
XI=2*ZR*ZI-.5*YI+QI
YR=ZR:YI=ZI:ZR=XR:ZI=XI
NEXT C
LOOP:NEXT V,H
DATA -.6,.2,.2,1.2,.56667,0
```

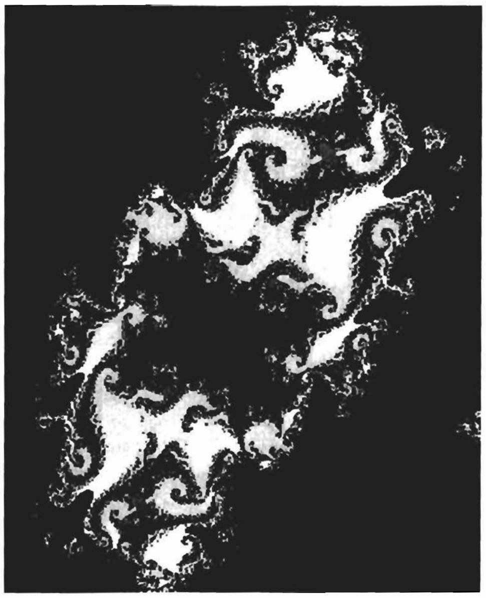


Figure 2
The Quartic picture

LISTING2

The floating-point registers in this program are:

```
FP7=XR   FP6=XI   FP5=YR   FP4=YI
FP3=ZR   FP2=ZI   FP1=ZZR  FP0=ZZI
```

Since most of the program's structure in Listing2 is the same as Listing1, I'll go right to SHOWIT and discuss the computations. A -1 is stored in d7, you'll see why later. The current X is stored in fp3 and Y in fp2. These values are squared in fp1 and fp0 and added together in fp7. This value is compared to 4 to see if the point is set or not.

New values are computed by multiplying ZR*ZI in fp6 and subtracting half the ZI value in fp2. Division is accomplished quickly in this case by using FSCALE #. If the number is negative the register is divided by 2 to that power (#-1=1/2); a positive value multiplies by 2 to that power (#2=4*). You can use the number directly (FSCALE #-1,fp5) or put the value in a data register as I did and then use FSCALE d7,fp5 - this method is quicker. After half of ZI is subtracted, QI is added to fp6. Then ZZR and ZZI are subtracted in fp7, half of ZR in fp3 subtracted, and QR added. These new values are stored in fp3 and fp2 as ZR and ZI.

Now ZR and ZI are squared in fp7 and fp6 and subtracted. Half of YR in fp5 is subtracted and QR added. ZR and ZI are multiplied in fp6 and doubled. Half of YI in fp4 is subtracted and QI added. Finally, fp3 and fp2 are moved to fp5 and fp4 as YR and YI; then fp7 and fp6 are moved to fp3 and fp2 as ZR and ZI.

As with the previous program you can select new values, change palettes, color-cycle, zoom, or quit at any time. This program is on the magazine disk* as QUARTICF.ASM and QUARTICF. If you make any changes, reassemble it using PHXASS QUARTICF.ASM and PHXLNK QUARTICF.O (PHXASS and PHXLNK are both on the magazine disk*). I hope you enjoy these two programs and that they whet your appetite for Mr. Pickover's books. Next time I'll discuss solving and graphing partial differential equations with the Amiga.

Editor's Note:
In the course of his article, Mr. Nee refers to several files and programs. In many cases, Mr. Senger refers to these programs as residing on the AC's TECH *companion disk. At press time, the* AC's TECH *disk is still in creation and we cannot be positive all the files he has requested will find room on the disk. Although every effort will be made to supply these files in a compressed format, we must retain room for the files and programs submitted by the remaining authors in this issue. If the files are not present, we will provide an additional readme file on the disk with suggestions on where the files may be obtained.* ➛

Listings

Listing1: QUATERF.ASM

```
;QUATERF.ASM
   section  text,code
   fpu   1
   bra   start
;Quaternions in double precision
   include  execmacros.i
   include  dosmacros.i
   include  intmacros.i
   include  gfxmacros.i
   include  dpmathmacros.i
   include  pmenu.i
depth     equ    5
count     equr  a2
across    equr  a3
down      equr  a4
;FP7  =  A0
;FP6  =  A1
;FP5  =  A2
;FP4  =  A3
;FP3  =  A0SQUARE
;FP2  =  A1SQUARE
;FP1  =  A2SQUARE
;FP0  =  A3SQUARE

@palette macro
   movea.l  vp(pc),a0
   lea      \1(pc),a1
   movea.l  gfxbase(pc),a6
   moveq    #32,d0
   jsr      loadrgb4(a6)
   endm
@newpalette macro
   lea      colortable(pc),a0
   lea      \1(pc),a1
   move.w   #15,d0
swap\@
   move.l   (a1)+,(a0)+
   dbra.s   d0,swap\@
   @palette colortable
   endm

@pset  macro         ;<across,down>
      move.w   \1,d0
      move.w   #399,d1        ;adjust down
      sub.w    \2,d1
      ext.l    d0
      ext.l    d1
      jsr      -324(a6)
      endm

start
 move.l sp,stack     ;save stack pointer

open_libs            ;open all the libraries we need
   openlib      int,done
   openlib      gfx,close_int
   openlib      dpmath,close_gfx
   openlib      dos,done
;  jmp          setup
options
   cls
   linefeed
   right    30
   boldface
   printstr 'Q U A T E R N I O N S'
   normal
   linefeed
   linefeed
   right    20
   printstr 'A quaternion Q = A0 + A1*i + A2*j + A3*k'
   linefeed
   linefeed
   right    20
   printstr 'This program displays Q = Q^2 + q  where -'
   linefeed
   linefeed
   right    20
   printstr 'A0 = x,     A1 = .05,  A2 = y,   A3 = .05'
   linefeed
   linefeed
   right    20
   printstr 'q0 = -.745, q1 = .113, q2 = .01, q3 = .01'
   linefeed
   linefeed
   right    20
   printstr 'Input: Xleft,   Xright,   Ybottom,   Ytop'
   linefeed
   linefeed
   right    20
   printstr '   A1/A3, q0, q1, q2, q3, minimum count'
   linefeed
   linefeed
   linefeed
   right    25
   style    italics,blue,white
   printstr 'Now, press the LMB to continue...'
   normal
   linefeed
option_loop
   btst     #6,$bfe001          ;LMB pressed?
   bne      option_loop

setup:                ;open a screen of 320 x 400
make_screen:
   openscreen  myscreen(pc),close_libs
make_window
   openwindow  mywindow(pc),close_screen
   mode        jam2
   @newpalette colortable2(pc)
   openmenu    menu0
dpf_quarternions_demo
msg_check
   cfm      msg_check
   cmpi.l   #menupick,d2
   bne.s    msg_check
   eval_menunumber
   tst.w    d0
   bne.s    msg_check
   tst.w    d1
   beq.s    coordinates
   cmpi.w   #1,d1
   beq.s    msg_check
   cmpi.w   #2,d1
   beq      quit
   bra.s    msg_check
coordinates
   removegadgets
   lea      gadget1buffer,a0
   bsr      convertdp
   movedp   xleft

   lea      gadget3buffer,a0
   bsr      convertdp
   movedp   xright

   lea      gadget4buffer,a0
   bsr      convertdp
   movedp   ybottom
   movedp   saveybottom

   lea      gadget2buffer,a0
   bsr      convertdp
   movedp   ytop

   lea      gadget5buffer,a0
   bsr      convertdp
   movedp   q0

   lea      gadget6buffer,a0
   bsr      convertdp
   movedp   q1

   lea      gadget7buffer,a0
   bsr      convertdp
   movedp   q2

   lea      gadget8buffer,a0
   bsr      convertdp
   movedp   q3

   lea      gadget9buffer,a0
   bsr      convertdp
   movedp   pzfive

   lea      gadget10buffer,a0
   bsr      convertdp
   absdp                       ;just to be sure
   fixdp
   move.w   d0,b
scale
   fltdp    320
   movedp   d0,d6
   subdp    xright,xleft
   movedp   d6,d2
   divdp
   movedp   xinc               ;x scale

   movedp   saveybottom,ybottom
   fltdp    400
   movedp   d0,d6
   subdp    ytop,ybottom
   movedp   d6,d2
   divdp
   movedp   yinc              ;y scale
   bra      showit

convertdp
   moveq.l  #0,d0
   moveq.l  #0,d1
   moveq.l  #0,d4    ;sign register
   moveq.l  #0,d5
   moveq.l  #0,d7    ;# characters right of decimal
   suba.l   a2,a2    ;clear a2; decimal flag register
   cmpi.b   #'-',(a0)   ;a minus sign?
   bne.s    positive
   bset     #31,d4   ;if so set last bit in d4
   addq.l   #1,a0    ;move over one space
positive
getdigit
   move.b   (a0)+,d5 ;next value to d5
   cmpi.b   #'.',d5  ;a decimal?
   bne.s    testdigit
   move.w   #1,a2    ;if so, flag a2
   clr.l    d7       ; and clear d7
   bra.s    getdigit
testdigit
   cmpi.b   #'9',d5     ;above '9' ?
   bhi.s    zero_check  ;branch if so
   cmpi.b   #'0',d5     ;below '0' ?
   blt.s    zero_check  ;branch if so
   andi.l   #$0f,d5     ;convert ASCII to decimal

   movedp   d0,d2       ;must multiply d0 * 10
   asl.l    #1,d1       ;d0 * 2
   roxl.l   #1,d0
   asl.l    #1,d3       ;d2 * 8
   roxl.l   #1,d2
   asl.l    #1,d3
   roxl.l   #1,d2
   asl.l    #1,d3
   roxl.l   #1,d2
   moveq.l  #0,d6
   add.l    d3,d1       ;d0 = d0 + d2
   addx.l   d2,d0
   add.l    d5,d1       ; + this digit
   addx.l   d6,d0

   addq.w   #1,d7       ;number of characters done
   cmpi.w   #16,d7      ;up to 15 ?
   bne      getdigit
dperror
   ;moveq    #1,d1      ;optional error flag
   ;rts
   bra      close_window   ;error - close everything!
zero_check
   movea.l  a0,a5       ;return string location
   tst.l    d1          ;don't try to convert 0
   bne.s    get_exponent   ;branch if not 0
   tst.l    d0          ;check second half
   beq.s    dp_done     ;branch if value is 0
get_exponent
   move.l   #$43f,d6    ;maximum exponent
1$
   subq.l   #1,d6       ;decrease exponent
   asl.l    #1,d1       ;d0 * 2
   roxl.l   #1,d0       ;rotate with carry from d1
   bcc.s    1$          ;branch if no carry
   moveq.l  #11,d5      ;move right 12 bits

shift_right
   lsr.l    #1,d0
   roxr.l   #1,d1
   dbra.s   d5,shift_right
adjust_exponent
   swap     d6          ;move to left word
   asl.l    #4,d6       ;move to left end
   or.l     d6,d0       ;put in d0
fraction_check
   cmpa.l   #0,a2       ;any decimal ?
   beq.s    do_sign     ;no
   subq.l   #1,d7       ;decrease number of digits
   bmi.s    do_sign     ;no digits after the decimal

decimal_adjust
   move.l   #$40240000,d2     ;dp 10
   moveq.l  #0,d3             ; part 2
   divdp
   dbra.s   d7,decimal_adjust ;do for all digits

do_sign
   or.l     d4,d0        ;add sign to d0
dp_done
   moveq.w  #0,d6        ;all is 'ok'
   rts
```

```
showit
   move.l   #1,alldone
   movea.l  rp(pc),a1
   movea.l  gfxbase(pc),a6
   pcls
   move.w   #0,down
11
   movedp   xleft,xx
   move.w   #0,across
12
   fmove.d  xx(pc),fp7
   fmove.d  ybottom(pc),fp5

   fmove.d  pzfive(pc),fp6
   fmove.d  pzfive(pc),fp4
   move.w   #0,count          ;clear the count
13
   fmove.x  fp7,fp3
   fmul.x   fp3,fp3     ;fp3 = a0sqr
   fmove.x  fp6,fp2
   fmul.x   fp2,fp2     ;fp2 = a1sqr
   fmove.x  fp5,fp1
   fmul.x   fp1,fp1     ;fp1 = a2sqr
   fmove.x  fp4,fp0
   fmul.x   fp0,fp0     ;fp0 = a3sqr

   fmove.d  fp0,variable   ;save fp0
   fadd.x   fp1,fp0
   fadd.x   fp2,fp0
   fadd.x   fp3,fp0     ;fp0=a0sqr+a1sqr+a2sqr+a3sqr
   fmove.l  fp0,d0      ;whole number in d0
   fmove.d  variable(pc),fp0  ;restore fp0
   cmpi.w   #4,d0             ;compare to 4
   bcs.s    continue          ;branch if lower
   cmpa.w   b,count           ;minimum count
   bcs      14                ;branch if lower
   move.w   count,d0          ;count -> color
   ;andi.l   #31,d0           ;optional color
   lsr.l    #2,d0             ;color
   foreground                 ;set apen
   @pset     across,down      ;pset the point
   bra      14
continue
   fadd.x   fp7,fp7     ;fp7 = 2 * aa0
   fmul.x   fp7,fp6     ;fp6 = 2 * aa0 * aa1
   fadd.d   q1(pc),fp6  ;fp6 = 2 * aa0 * aa1 + q1

   fmul.x   fp7,fp5     ;fp5 = 2 * aa0 * aa2
   fadd.d   q2(pc),fp5  ;fp5 = 2 * aa0 * aa1 + q2

   fmul.x   fp7,fp4     ;fp4 = 2 * aa0 * aa3
   fadd.d   q3(pc),fp4  ;fp4 = 2 * aa0 * aa3 + q3

   fsub.x   fp2,fp3     ;fp3 = a0sqr - a1sqr
   fsub.x   fp1,fp3     ; - a2sqr
   fsub.x   fp0,fp3     ; - a3sqr
   fadd.d   q0(pc),fp3  ; + q0
   fmove.x  fp3,fp7     ;new aa0

   add.w    #1,count    ;increase count
   cmpa.w   #127,count  ;up to maximum yet ?
   bne      13          ;branch if not
14
   adddp    xx,xinc     ;increase xx by xscale
   movedp   xx
check_for_message
   cfm      no_message
   cmpi.l   #menupick,d2
   beq      check_menu
   cmpi.l   #mousebuttons,d2
   beq      zoom
   tst.w    d3
   beq      no_message
   andi.w   #$ffff,d3
   cmpi.w   #$50,d3           ;F1
   beq.s    do_palette1
   cmpi.w   #$51,d3           ;F2
   beq.s    do_palette2
   cmpi.w   #$52,d3           ;F3
   beq.s    do_palette3
   cmpi.w   #$53,d3           ;F4
   beq      do_palette4
   cmpi.w   #$54,d3           ;F5
   beq      do_palette5
   cmpi.w   #$55,d3           ;F6
   beq      do_palette6
   cmpi.w   #$56,d3           ;F7
   beq      do_palette7
   cmpi.w   #$57,d3           ;F8
   beq      do_palette8
   cmpi.w   #$58,d3           ;F9
   beq      do_palette9
   cmpi.w   #$59,d3           ;F10
   beq      do_palette0
   cmpi.w   #$4c,d3           ;up-arrow
   beq      colorcycleup
   cmpi.w   #$4d,d3
   beq      colorcycledown    ;down-arrow
   bra      no_message

do_palette1
   @newpalette colortable1
   bra      no_message
do_palette2
   @newpalette colortable2
   bra      no_message
do_palette3
   @newpalette colortable3
   bra      no_message
do_palette4
   @newpalette colortable4
   bra      no_message
do_palette5
   @newpalette colortable5
   bra      no_message
do_palette6
   @newpalette colortable6
   bra      no_message
do_palette7
   @newpalette colortable7
   bra.s    no_message
do_palette8
   @newpalette colortable8
   bra.s    no_message
do_palette9
   @newpalette colortable9
   bra.s    no_message
do_palette0
   @newpalette colortable0
   bra.s    no_message

colorcycleup
   lea      colortable(pc),a0
   move.w   2(a0),d2          ;save #1
   move.w   #2,d1
   move.w   #29,d0            ;28-0=29 changes
cycleup
   move.w   2(a0,d1.w),0(a0,d1.w)   ;move up one
   addq.w   #2,d1             ;next color
   dbra.s   d0,cycleup
   move.w   d2,62(a0)         ;now #31
usecolors
   @palette colortable
   tst.l    alldone
   beq      now_what
   bra.s    no_message

colorcycledown
   lea      colortable(pc),a0
   move.w   62(a0),d2
   move.w   #60,d1
   move.w   #29,d0
cycledown
   move.w   0(a0,d1.w),2(a0,d1.w)
   subq.w   #2,d1
   dbra.s   d0,cycledown
   move.w   d2,2(a0)
   bra.s    usecolors

check_menu
   eval_menunumber
   tst.w    d0                ;menu0 ?
   bne.s    no_message        ;no
   cmpi.w   #1,d1             ;item1 - coordinates
   beq.s    do_coordinates
   cmpi.w   #2,d1             ;item2 - quit
   beq      quit
   bra      no_message

do_coordinates
   pcls
   closemenu
   closewindow
   bra      make_window

zoom
   andi.l   #$0000ffff,d5  ;mouse x
   andi.l   #$0000ffff,d6  ;mouse y
   move.l   d5,startx
   move.l   d6,starty
   mode     complement
lmb_down
   cfm      lmb1
   cmpi.w   #selectup,d3
   beq      lmb_up
lmb1
   andi.l   #$0000ffff,d5  ;mouse x
   andi.l   #$0000ffff,d6  ;mouse y
   move.l   d5,endx
   move.l   d6,endy
```

```
   box      startx,starty,endx,endy,21
   box      startx,starty,endx,endy,21
   bra      lmb_down
lmb_up
   box      startx,starty,endx,endy,21
   mode     jam1
new_coordinates
check_them
   move.l   startx,d0
   move.l   endx,d1
   cmp.l    d1,d0
   blo.s    nc1          ;startx < endx
   exg      d1,d0
   move.l   d0,startx
   move.l   d1,endx
nc1
   move.l   starty,d0
   move.l   endy,d1
   cmp.l    d1,d0
   blo.s    nc2          ;starty < endy
   exg      d0,d1
   move.l   d0,starty
   move.l   d1,endy

nc2
   move.l   startx,d0
   fltdp
   movedp   xinc,d2
   muldp
   adddp    xleft
   movedp   newxleft
   move.l   endx,d0
   fltdp
   movedp   xinc,d2
   muldp
   adddp    xleft
   movedp   xright
   movedp   newxleft,xleft

   move.l   starty,d0
   fltdp
   movedp   yinc,d2
   muldp
   movedp   d0,d2
   movedp   ytop,d0
   subdp
   movedp   newytop
   move.l   endy,d0
   fltdp
   movedp   yinc,d2
   muldp
   movedp   d0,d2
   movedp   ytop,d0
   subdp
   movedp   ybottom
   movedp   saveybottom
   movedp   newytop,ytop

   move.w   b,d0
   addq.w   #5,d0
   move.w   d0,b         ;increase the minimum count
   bra      scale

no_message
   add.w    #1,across      ;across one space
   cmpa.w   #320,across    ;all way across yet ?
   bne      12             ;branch if not

   adddp    ybottom,yinc   ;increase yy by yscale
   movedp   ybottom
   add.w    #1,down        ;down one space
   cmpa.w   #400,down      ;all way down yet ?
   bne      11             ;branch if not
   move.l   #0,alldone     ;set finished flag

now_what
   cfm      now_what_done
   cmpi.l   #menupick,d2
   beq      check_menu1
   cmpi.l   #mousebuttons,d2
   beq      zoom
   tst.l    d3
   beq      now_what_done
   andi.l   #$0000ffff,d3
   cmpi.w   #$50,d3        ;F1
   beq.s    do_palette11
   cmpi.w   #$51,d3        ;F2
   beq      do_palette21
   cmpi.w   #$52,d3        ;F3
   beq      do_palette31
   cmpi.w   #$53,d3        ;F4
   beq      do_palette41
   cmpi.w   #$54,d3        ;F5
   beq      do_palette51
   cmpi.w   #$55,d3        ;F6
   beq      do_palette61
   cmpi.w   #$56,d3        ;F7
   beq      do_palette71
   cmpi.w   #$57,d3        ;F8
   beq      do_palette81
   cmpi.w   #$58,d3        ;F9
   beq      do_palette91
   cmpi.w   #$59,d3        ;F10
   beq      do_palette01
   cmpi.w   #$4c,d3        ;up-arrow
   beq      colorcycleup
   cmpi.w   #$4d,d3        ;down-arrow
   beq      colorcycledown
now_what_done
   bra      now_what

do_palette11
   @newpalette  colortable1
   bra      now_what
do_palette21
   @newpalette  colortable2
   bra      now_what
do_palette31
   @newpalette  colortable3
   bra      now_what
do_palette41
   @newpalette  colortable4
   bra      now_what
do_palette51
   @newpalette  colortable5
   bra      now_what
do_palette61
   @newpalette  colortable6
   bra      now_what
do_palette71
   @newpalette  colortable7
   bra      now_what
do_palette81
   @newpalette  colortable8
   bra      now_what
do_palette91
   @newpalette  colortable9
   bra      now_what
do_palette01
   @newpalette  colortable0
   bra      now_what

check_menu1
   eval_menunumber
   tst.w    d0
   bne      now_what
   cmpi.w   #1,d1
   beq      do_coordinates
   cmpi.w   #2,d1
   beq.s    quit
   bra      now_what

quit
close_window
   closewindow
close_screen
   closescreen
close_libs
   closelib dpmath
close_gfx
   closelib gfx
close_int
   closelib int
done
   move.l   stack(pc),sp
   rts

   evenpc

stack       dc.l  0        ;reserve storage locations
gfxbase     dc.l  0
intbase     dc.l  0
dosbase     dc.l  0
dpmathbase  dc.l  0
                           ;library names
gfx         dc.b  'graphics.library',0
   evenpc
int         dc.b  'intuition.library',0
   evenpc
dos         dc.b  'dos.library',0
   even
dpmath      dc.b  'mathieeedoubbas.library',0
   evenpc

myscreen:
   dc.w 0,0,320,400,depth  ;depth is 5
   dc.b 0,1
   dc.w $4
   dc.w customscreen
   dc.l 0,0,0,0
```

```
mywindow
   dc.w 0,0,320,400
   dc.b 0,1
   dc.l mousebuttons!menupick!mousemove!rawkey
   dc.l activate!smartrefresh!borderless!mousereport
   dc.l gadget1,0
   dc.l 0
   dc.l 0,0
   dc.w 0,0,0,0
   dc.w customscreen

   evenpc
xleft     dc.l  0,0      ;reserve space for both parts
newxleft dc.l  0,0
xright    dc.l  0,0
xx        dc.l  0,0
xinc      dc.l  0,0
ybottom   dc.l  0,0
saveybottom dc.l  0,0
ytop      dc.l  0,0
newytop   dc.l  0,0
yinc      dc.l  0,0
q0        dc.l  0,0
q1        dc.l  0,0
q2        dc.l  0,0
q3        dc.l  0,0
pzfive    dc.l  0,0
variable dc.l  0,0      ;variable
startx    dc.l  0
endx      dc.l  0
starty    dc.l  0
endy      dc.l  0
alldone   dc.l  0
b         dc.w  0
   even
colortable1
   dc.w  $000,$fff,$f0f,$e0f,$d1f,$c1f,$a1f,$92f
   dc.w  $62f,$42f,$20f,$00f,$02d,$03b,$059,$078
   dc.w  $096,$0b4,$0f0,$3f0,$7f0,$bf0,$ff0,$fe0
   dc.w  $fd0,$fc0,$fa0,$f80,$f60,$f40,$f20,$f00
   even
colortable2
   dc.w  $000,$fae,$fca,$fd9,$fe6,$ee6,$ed5,$ec2
   dc.w  $eb0,$eb2,$eb3,$ea5,$fa6,$f53,$f21,$f00
   dc.w  $855,$0f0,$880,$faa,$855,$0f0,$880,$faa
   dc.w  $d0e,$95f,$6af,$358,$00f,$40a,$808,$c04
   even
colortable3
   dc.w  $000,$100,$200,$300,$400,$500,$600,$700
   dc.w  $800,$900,$a00,$b00,$c00,$d00,$d80,$e00
   dc.w  $e10,$e20,$e30,$e40,$e50,$e60,$e70,$f70
   dc.w  $f80,$f90,$fa0,$fb0,$fc0,$fd0,$fe0,$ff0
   even
colortable4
   dc.w  $620,$ff0,$ff2,$de2,$fd2,$fc1,$fa1,$f91
   dc.w  $f81,$f71,$f51,$f40,$f30,$f20,$f10,$f00
   dc.w  $620,$630,$740,$760,$870,$820,$790,$700
   dc.w  $6a0,$580,$5b0,$3c0,$0c0,$5d5,$aea,$fff
   even
colortable5
   dc.w  $000,$f00,$e20,$d40,$c60,$b80,$aa0,$8b0
   dc.w  $6c0,$4d0,$2e0,$0f0,$0e2,$0d4,$0c6,$0b8
   dc.w  $0aa,$08b,$06c,$04d,$02e,$00f,$00e,$00d
   dc.w  $00c,$00b,$00a,$009,$007,$005,$003,$001
   even
colortable6
   dc.w  $000,$400,$501,$720,$710,$821,$831,$841
   dc.w  $951,$a61,$a71,$a81,$b91,$ba1,$bb2,$cc2
   dc.w  $ac2,$9c2,$7c2,$6c2,$4d2,$2d1,$1d2,$1d4
   dc.w  $1e5,$1e7,$1ea,$1ec,$1ee,$0df,$0bf,$09f
   even
colortable7
   dc.w  $000,$620,$730,$951,$a72,$c93,$db4,$fd5
   dc.w  $ee4,$cd3,$9c2,$7a2,$591,$481,$270,$150
   dc.w  $062,$074,$067,$047,$008,$308,$709,$907
   dc.w  $a05,$a13,$b11,$b43,$c72,$da3,$dd3,$be4
   even
colortable8
   dc.w  $000,$fff,$ecd,$c9c,$b7a,$a59,$837,$726
   dc.w  $559,$66a,$77b,$99b,$aac,$ccd,$dde,$fff
   dc.w  $ded,$cdb,$bc9,$9b7,$8a6,$794,$685,$572
   dc.w  $58a,$69b,$7ab,$9bc,$acd,$bcc,$cee,$fff
   even
colortable9
   dc.w  $000,$fe0,$db0,$c90,$a60,$950,$730,$620
   dc.w  $fe0,$af0,$3f0,$0f3,$0fa,$0df,$07f,$00f
   dc.w  $fe0,$eb0,$c90,$b90,$a60,$940,$730,$620
   dc.w  $fe0,$de0,$ad0,$7c0,$5b0,$3a0,$270,$080
   even
colortable0
   dc.w  $000,$004,$008,$00b,$00f,$00b,$008,$004
   dc.w  $000,$430,$770,$ba0,$fe0,$ba0,$860,$430
   dc.w  $000,$300,$700,$a00,$e00,$a00,$700,$300
   dc.w  $000,$020,$040,$060,$080,$060,$040,$020
   even
```

```
colortable
   dc.w  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   dc.w  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
   even
menus
   makemenu menu0,'Quarternion',,0,1
    makeitem menu0item0,'Draw',menu0item1,0,$153,$2
    makeitem menu0item1,'Coordinates',menu0item2,10,$53,$1
    makeitem menu0item2,'Quit',,20,$52
gadgets
   makestrgadget gadget1,'Xleft',gadget2,40,100,1
   makestrgadget gadget2,'Ytop',gadget3,110,75,2
   makestrgadget gadget3,'Xright',gadget4,180,100,3
   makestrgadget gadget4,'Ybot',gadget5,110,125,4
   makestrgadget gadget5,'Q0',gadget6,55,160,5
   makestrgadget gadget6,'Q1',gadget7,170,160,6
   makestrgadget gadget7,'Q2',gadget8,55,210,7
   makestrgadget gadget8,'Q3',gadget9,170,210,8
   makestrgadget gadget9,'A1 & A3',gadget10,55,260,9
   makestrgadget gadget10,'Min Count',,170,260,10
   even
gadget1buffer  dc.b  '-1.500000',0     ;xleft
   even
gadget2buffer  dc.b  '.800000',0       ;ytop
   even
gadget3buffer  dc.b  '1.500000',0      ;xright
   even
gadget4buffer  dc.b  '-.800000',0      ;ybottom
   even
gadget5buffer  dc.b  '-.745000',0      ;q0
   even
gadget6buffer  dc.b  '.113000',0       ;q1
   even
gadget7buffer  dc.b  '.010000',0       ;q2
   even
gadget8buffer  dc.b  '.010000',0       ;q3
   even
gadget9buffer  dc.b  '.050000',0 ;constant (aa1 & aa3)
   even
gadget10buffer dc.b  '12',0      ;minimum color count
   even
   end
;try these coordinates
;  -1.5,1.5,-.8,.8,-.745,.113,.01,.01,.05  entire Q    <12>
;  -.1,1.4,-.6,.6,-.745,.113,.01,.01,.05   right side  <23>
;  -1.45,-1,-.2,.2,-.745,.113,.01,.01,.05  left side   <17>
;  -.6,.6,.1,.9,-.745,.113,.01,.01,.05     top         <15>
```

Listing2: QUARTICF.ASM

```
;QUARTICF.ASM
   section  text,code
   fpu   1
   bra   start
;Quartic Variations in double precision, floating
   include  execmacros.i
   include  dosmacros.i
   include  intmacros.i
   include  gfxmacros.i
   include  dpmathmacros.i
   include  pmenu.i
depth    equ   5
count    equr  a2
across   equr  a3
down     equr  a4

;FP7 = XR
;FP6 = XI
;FP5 = YR
;FP4 = YI
;FP3 = ZR
;FP2 = ZI
;FP1 = ZZR
;FP0 = ZZI

@palette macro
   movea.l  vp(pc),a0
   lea      \1(pc),a1
   movea.l  gfxbase(pc),a6
   moveq    #32,d0
   jsr      loadrgb4(a6)
   endm
@newpalette macro
   lea      colortable(pc),a0
   lea      \1(pc),a1
   move.w   #15,d0
swap\@
   move.l   (a1)+,(a0)+
   dbra.s   d0,swap\@
   @palette colortable
   endm

@pset  macro          ;<across,down>
      move.w   \1,d0
      move.w   #399,d1        ;adjust down
      sub.w    \2,d1
      ext.l    d0
      ext.l    d1
      jsr      -324(a6)
      endm

start
 move.l sp,stack               ;save stack pointer

open_libs                      ;open all the libraries we need
   openlib     int,done
   openlib     gfx,close_int
   openlib     dpmath,close_gfx
   openlib     dos,done
;  jmp         setup
options
   cls
   linefeed
   right    20
   boldface
   printstr 'Q U A R T I C   V A R I A T I O N S'
   normal
   linefeed
   linefeed
   right    22
   printstr 'Zcomplex = X^2 + Y^2 + QR + QI'
   linefeed
   linefeed
   right    18
   printstr 'Xcomplex = Zreal^2 + Zimag^2 + QR + QI'
   linefeed
   linefeed
   right    16
   printstr 'Ycomplex = Zcomplex  , Zcomplex = Xcomplex'
   linefeed
   linefeed
   right    26
   printstr 'QR = .56667    QI = 0'
   linefeed
   linefeed
   right    16
   printstr 'Input: Xleft,   Xright,   Ybottom,   Ytop'
   linefeed
   linefeed
   right    16
   printstr '       QR,      QI,         minimum count'
   linefeed
   linefeed
   linefeed
   right    21
   style    italics,blue,white
   printstr 'Now, press the LMB to continue...'
   normal
   linefeed
option_loop
   btst     #6,$bfe001     ;LMB pressed?
   bne      option_loop

setup:                   ;open a screen of 320 x 400
make_screen:
   openscreen  myscreen(pc),close_libs
make_window
   openwindow  mywindow(pc),close_screen
   mode        jam2
   @newpalette colortable2(pc)
   openmenu    menu0
dpf_quartic_demo
msg_check
   cfm      msg_check
   cmpi.l   #menupick,d2
   bne.s    msg_check
   eval_menunumber
   tst.w    d0
   bne.s    msg_check
   tst.w    d1
   beq.s    coordinates
   cmpi.w   #1,d1
   beq.s    msg_check
   cmpi.w   #2,d1
   beq      quit
   bra.s    msg_check
coordinates
   removegadgets
   lea      gadget1buffer,a0
   bsr      convertdp
   movedp   xleft

   lea      gadget3buffer,a0
   bsr      convertdp
   movedp   xright

   lea      gadget4buffer,a0
   bsr      convertdp
   movedp   ybottom
   movedp   saveybottom

   lea      gadget2buffer,a0
   bsr      convertdp
   movedp   ytop

   lea      gadget5buffer,a0
   bsr      convertdp
   movedp   qr

   lea      gadget6buffer,a0
   bsr      convertdp
   movedp   qi

   lea      gadget7buffer,a0
   bsr      convertdp
   absdp                    ;just to be sure
   fixdp
   move.w   d0,b
scale
   fltdp    320
   movedp   d0,d6
   subdp    xright,xleft
   movedp   d6,d2
   divdp
   movedp   xinc            ;x scale

   movedp   saveybottom,ybottom
   fltdp    400
   movedp   d0,d6
   subdp    ytop,ybottom
   movedp   d6,d2
   divdp
   movedp   yinc            ;y scale
   bra      showit

convertdp
   moveq.l  #0,d0
   moveq.l  #0,d1
   moveq.l  #0,d4       ;sign register
   moveq.l  #0,d5
   moveq.l  #0,d7    ;# characters right of decimal
   suba.l   a2,a2    ;clear a2; decimal flag register
   cmpi.b   #'-',(a0)    ;a minus sign?
   bne.s    positive
   bset     #31,d4      ;if so set last bit in d4
   addq.l   #1,a0       ;move over one space
positive
getdigit
   move.b   (a0)+,d5    ;next value to d5
   cmpi.b   #'.',d5     ;a decimal?
   bne.s    testdigit
   move.w   #1,a2       ;if so, flag a2
   clr.l    d7          ; and clear d7
   bra.s    getdigit
testdigit
   cmpi.b   #'9',d5     ;above '9' ?
   bhi.s    zero_check  ;branch if so
   cmpi.b   #'0',d5     ;below '0' ?
   blt.s    zero_check  ;branch if so
   andi.l   #$0f,d5     ;convert ASCII to decimal

   movedp   d0,d2       ;must multiply d0 * 10
   asl.l    #1,d1       ;d0 * 2
   roxl.l   #1,d0
   asl.l    #1,d3       ;d2 * 8
   roxl.l   #1,d2
   asl.l    #1,d3
   roxl.l   #1,d2
   asl.l    #1,d3
   roxl.l   #1,d2
   moveq.l  #0,d6
   add.l    d3,d1       ;d0 = d0 + d2
   addx.l   d2,d0
   add.l    d5,d1       ; + this digit
   addx.l   d6,d0

   addq.w   #1,d7       ;number of characters done
   cmpi.w   #16,d7      ;up to 15 ?
   bne      getdigit
dperror
   ;moveq    #1,d1      ;optional error flag
   ;rts
   bra      close_window   ;error - close everything!
zero_check
   movea.l  a0,a5       ;return string location
   tst.l    d1          ;don't try to convert 0
   bne.s    get_exponent   ;branch if not 0
   tst.l    d0          ;check second half
   beq.s    dp_done     ;branch if value is 0
get_exponent
   move.l   #$43f,d6    ;maximum exponent
1$
   subq.l   #1,d6       ;decrease exponent
   asl.l    #1,d1       ;d0 * 2
   roxl.l   #1,d0       ;rotate with carry from d1
   bcc.s    1$          ;branch if no carry
   moveq.l  #11,d5      ;move right 12 bits

shift_right
   lsr.l    #1,d0
   roxr.l   #1,d1
   dbra.s   d5,shift_right
adjust_exponent
   swap     d6          ;move to left word
   asl.l    #4,d6       ;move to left end
   or.l     d6,d0       ;put in d0
fraction_check
   cmpa.l   #0,a2       ;any decimal ?
   beq.s    do_sign     ;no
   subq.l   #1,d7       ;decrease number of digits
   bmi.s    do_sign     ;no digits after the decimal

decimal_adjust
   move.l   #$40240000,d2  ;dp 10
   moveq.l  #0,d3          ; part 2
   divdp
   dbra.s   d7,decimal_adjust ;do for all digits

do_sign
   or.l     d4,d0       ;add sign to d0
dp_done
   moveq.w  #0,d6       ;all is 'ok'
   rts

showit
   move.l   #-1,d7
   move.l   #1,alldone
   movea.l  rp(pc),a1
   movea.l  gfxbase(pc),a6
   pcls
   move.w   #0,down
11
   movedp   xleft,xx
   move.w   #0,across
12
   fmove.d  xx(pc),fp3
   fmove.d  ybottom(pc),fp2
   fmove.d  #0,fp6
   fmove.x  fp6,fp5
   fmove.x  fp6,fp4
   move.w   #0,count    ;clear the count
13
   fmove.x  fp3,fp1
   fmul.x   fp1,fp1     ;fp1 = zr*zr
   fmove.x  fp2,fp0
   fmul.x   fp0,fp0     ;fp0 = zi*zi
   fmove.x  fp1,fp7
   fadd.x   fp0,fp7     ;fp7 = zr*zr + zi*zi
   fmove.l  fp7,d0      ;whole number in d0
   cmpi.w   #4,d0       ;compare to 4
   bcs.s    continue    ;branch if lower
   cmpa.w   b,count     ;minimum count
   bcs      14          ;branch if lower
   move.w   count,d0    ;count -> color
   ;andi.l   #31,d0     ;optional color
   lsr.l    #2,d0       ;color
   foreground           ;set apen
   @pset     across,down   ;pset the point
   bra      14
continue
   fmove.x  fp3,fp6     ;fp6 = zr
   fmul.x   fp2,fp6     ;fp6 = zr*zi
   fadd.x   fp6,fp6     ;fp6 = 2*zr*zi
   fscale.b d7,fp2      ;fp2 = .5*zi
   fsub.x   fp2,fp6     ;fp6 = 2*zr*zi - .5*zi
   fadd.d   qi(pc),fp6  ;fp6 = 2*zr*zi - .5*zi + qi
```

This is not a complete listing:

The balance of this program as well as additional marcros, etc. can be found on the companion *AC's TECH* disk for this issue.

✔

Please Write to:
Bill Nee
c/o AC's TECH
P.O. Box 2140
Fall River, MA 02722-2140

AC's Back Issue Index

Amazing Computing

❦ Vol.8, No.5, May 1993
Highlights Include:
"Directory Opus", review of the latest version of Directory Opus and a start-up tutorial by Merrill Callaway
"Media Madness," explores the inside of Blue Ribbon Soundwork's new Media Madness, by Todor Fay & David Miller
"SuperJAM 1.1," a review of the latest release of SuperJAM! by Rick Manasa
"ImageFX," review by R. Shamms Mortier
ALSO: Super VideoSlot for May—The New Graphics Modes!

❦ Vol.8, No.6, June 1993
Highlights Include:
"AMOS Turns Professional",review of a major upgrade hailed as a comprehensive development system, by Jimmy Rose
"Searching Medical Literature," using the Amiga to tap the vast resources of medical on-line services, by Dr. Michael Tobin
ALSO: Newsletter Design, ARexx Programming, Hot Diversions

❦ Vol.8, No.7, July 1993
Highlights Include:
"TypeSMITH 1.0", review of Soft-Logik's new font editor, by Merrill Callaway
"OpalPaint 2.0," review of the latest version of this paint program for the OpalVision board, by R. Shamms Mortier
"Structured Drawing," basic features and advanced techniques, by Dan Weiss
"DeluxePaint IV AGA," review of the latest paint package for the AGA machines, by R. Shamms Mortier
ALSO: Super VideoSlot, ARexx, and New Products!

❦ Vol.8, No 8, August 1993
Highlights Include:
"Amiga Vision Professional", review Commodore's upgraded authoring system, by Douglas J. Nakakihara
"Art Department Professional 2.3," review of the latest release of AdPro from ASDG, by Merrill Callaway
"Professional Page 4.0," the latest incarnation of Pro Page, by Rick Manasa
"Pseudo Radiosity Effects," why ray tracing is not an accurate model of true light behavior, by Mark Hoffman
"T-Rexx Professional", a review of the latest release of T-Rexx from ASDG, by Merrill Callaway
ALSO: AC Phone Book: A directory of Amiga Developers!

❦ Vol.8, No 9, September 1993
Highlights Include:
"Adventures with Aladdin",Part III of this tutorial series on Aladdin 4D, by R. Shamms Mortier
"CanDo,"First installment of this series for CanDo programmers, by Randy Finch
"Caligari 24," Review of version 3.0 of this 24-bit software, by R. Shamms Mortier
"Coming Attractions," A look into the future attractions in Amiga games, by Henning Vahlenkamp
ALSO: WOCA—Australia & Summer CES!

❦ Vol.8, No 10, October 1993
Highlights Include:
"Making Waves", Focus on the wave requester in Part IV of the Aladdin series, R. Shamms Mortier
"Clouds in Motion," Animated clouds in Scenery Animator, by R. Shamms Mortier
"Media Madness," Discover what it can do for Bars&Pipes, by Rick Manasa
"Bars&Pipes Professional 2.0," review by Rick Manasa
"Bernoulli MultiDisk 150", A review of this great Iomega drive.
ALSO: Commodore's new CD32!

❦ Vol.8, No 11, November 1993
Highlights Include:
"CanDo", This installment covers developing a custom object by combining several standard CanDo objects, by Randy Finch.
"Brilliance," A complete review of this hot new paint and animation program from Digital Creations, by Frank McMahon.
"Online," The introduction of this new telecommunications column for the Amiga, by Rob Hays.
"Get Graphic: Digital Image F/X," The introduction of AC's new graphics column, by William Frawley.
"Picasso II", A review of one of the best new graphics cards available, by Mark Ricken.
ALSO: WOCA Pasadena: Commodore introduces CD-32! Plus, the incredible LightRave, a Video Toaster emulator!

❦ Vol.9, No 1, January 1994
Highlights Include:
"Designing Holiday Cards", Using your favorite DTP programs to create holiday cards, by Dan Weiss.
"Accent on Multimedia," First in a series exploring the history and concepts behind multimedia, by R. Shamms Mortier.
"Primera Printer," Review of this low end, inexpensive color printer, by Merrill Callaway.
"Commodore 1942 Monitor," In-depth study of this comprehensive Amiga paint package, by R. Shamms Mortier.
ALSO: Commodore Shareholders Movement

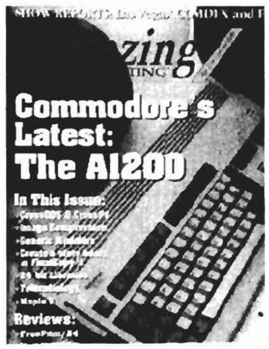
❦ Vol.9, No 2, February 1994
Highlights Include:
"Amiga on Internet", Exploration of Internet and its services, by Henning Vahlenkamp.
"EGS 28/24 Spectrum," A review of this hot graphics card from GVP, by Mark Hoffman.
"Magic Lantern" A new animation compiling program for all Amiga display modes, by R. Shamms Mortier.
"Get Graphic: Digital Image F/X," Using ARexx, Opal Paint, ADPro, and DeluxePaint to process images, by William Frawley.
ALSO: Exclusive interview with Lew Eggebrecht!

❦ Vol.9, No 3, March 1994
Highlights Include:
"Amiga Stars at Medical Convention", Medical multimedia on the Amiga, by Michael Tobin, M.D..
"CanDo vs. HELM," Head-to-head review of two leading Amiga authoring systems, by Randy Finch.
"PD Update," This month, a description of *AlertPatch 2.9* and other shareware and freeware utilities, by Henning Vahlenkamp.
"Scala MM300," A review of the program believed to be "hot stuff" for anyone doing interactive media work, by R. Shamms Mortier.
ALSO: And furthermore: The Amiga takes the stage in the Broadway production of The Who's Tommy!

❦ Vol.9, No 4, April 1994
Highlights Include:
"Computer Cafe Serves Up Shasta", The design team at Computer Cafe creates incredible "can-a-mation" for a beverage commercial, by Robert Van Buren.
"Aladdin 4D Review," Comprehensive look at the latest version of Aladdin, by R. Shamms Mortier.
"AGA Chipset and the Amiga: CD^{32} to the Rescue!" What does the future hold for CD32 and Amiga games? Jeff James has the inside scoop, by Jeff James.
"Sync Tips," Video returns to the pages of AC, featuring Oran Sands.
ALSO: Exclusive interview with renowned Amiga artist Jim Sachs.

❦ Vol.9, No 5, May 1994
Highlights Include:
"Desktop Publishing for Profit", Resume design: A simple and profitable way to break into the desktop publishing field, by Dan Weiss.
"24-bit Painting Techniques," Innovative tips and tricks anyone can use to make their computer paintings look better, by Mark Hoffman.
"PD Update," This month, *MegaBall 3.0, Motorola Invaders, New World,* and more, by Henning Vahlenkamp.
"MicroBotics MBX-1200Z," A review of this handy math coprocessor and 32-bit RAM add-on card for the Amiga 1200, by Rob Hays.
ALSO: The long-awaited Amiga 4000 Tower is showcased at the Cebit show in Germany.

❦ Vol.9, No.6, June 1994
Highlights Include:
"CanDo," Select, enter, and play music files, by Randy Finch.
"NAB show report," *AC* travels to Las Vegas for the latest releases and announcements.
"Making an Article Database," Create a simple database to keep track of magazine articles using the HELM authoring system, by Doug Nakakihara.
"A Survival Guide to CD-ROM Part I," The first in a four part series designed to take the confusion out of CD-ROM devices, by Mark Rickan.
"Bubbles vs Heat," Fargo's Primera Color Printer & Canon's BJC-600, by Dwinn Craig.
"1994 Reader's Choice Awards Ballot".
"TypeSmith 2.0," Review, by Merrill Callaway.
"The A 64 Package 3.0," This new release brings quality C64 emulation to the Amiga, by Henning Vahlenkamp.
"MIDIquest 4.5 & TECHquest," Review, by Shamms Mortier.



AC's TECH

❦ AC's TECH, Vol. 3, No. 2
Highlights Include:
"Olé," An arcade game programmed in AMOS BASIC, by Thomas J. Eshelman.
"Programming the Amiga in Assembly Language Part 6," by William Nee
"Wrapped Up with True BASIC," Text and Graphics wrapping modules in True BASIC, by Dr. Roy M. Nuzzo
"ARexx Disk Cataloger," An AmigaDOS manipulator that produces a text file containing information about the floppy disks you want cataloged, by T. Darrel Westbrook
AND LOTS MORE ON DISK!

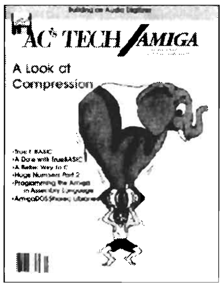
❦ AC's TECH, Vol. 3, No.3
Highlights Include:
"Rexx Rainbow Library," A review by Merrill Callaway
"All You Ever Wanted to Know About Morphing," An in-depth look at morphing for Imagine by Bruno Costa and Lucia Darsa
"Custom 3D Graphics Package Part I," Designing a custom 3D graphics package by Laura Morisson.
"Build a Second Joystick Port," A simple hardware project for an additional joystick port by Jaques Hallee.
AND LOTS MORE ON DISK!

❦ AC's TECH, Vol. 3, No. 4
Highlights Include:
"Custom 3D Graphics Package Part II," Put the finishing touches on your own graphics package by Laura Morisson.
"TruBASIC Input Mask," An interesting TrueBASIC utility by T. Darrell Westbrook.
"Time Efficient Animations," Make up for lost time with this great animation utility by Robert Galka.
"F-BASIC 5.0," A review of this latest version of F-BASIC by Jeff Stein.
PLUS: CD32 Development Info!

❦ AC's TECH, Vol. 4, No. 1
Highlights Include:
"Artificial Life," Artificial life, intelligence and other technical tidbits in this piece, by John Iovine.
"Huge Numbers Part I," Creative number crunching, by Michael Greibling.
"Pseudo-random Number Generation," Generating sequences of random numbers—almost, by Cristopher Jennings.
"Draw 5.0," Door prize selection in AMOS Professional, by T. Darrell Westbrook.
"Programming the Amiga in Assembly Language," Complex functions are explored, by William P. Nee.
"Writing a Function Genie for Pro Draw," Create a calendar beginning October 1582, by Keith D. Brown.

❦ AC's TECH, Vol. 4, No. 2
Highlights Include:
"True F-BASIC," What do you get when you cross True BASIC with F-BASIC? You'll be surprised, by Roy M. Nuzzo.
"Huge Numbers Part II," Creative number crunching, by Michael Greibling.
"Building an Audio Digitizer," Create a simple audio digitizer for your Amiga, by John Iovine.
"A Look at Compression," Various compression techniques and what they do for you, by Dan Weiss.
"Programming the Amiga in Assembly Language," Using the math coprocessor, by William P. Nee.
"AmigaDOS Shared Libraries," Examining AmigaDOS libraries and their functions, by Daniel Stenberg.

1-800-345-3360



BACK ISSUE SPECIALS! SEE PAGE 32 FOR DETAILS

Complete selection of Amazing Computing and AC's TECH AVAILABLE!

WHAT HAVE YOU BEEN MISSING? Have you missed information on how to add ports to your Amiga for under $70, how to work around *DeluxePaint*'s lack of HAM support, how to deal with service bureaus, or how to put your Super 8 films on video tape, along with Amiga graphics? Do you know the differences among the big three DTP programs for the Amiga? Does the ARexx interface still puzzle you? Do you know when it's better to you use the CLI? Would you like to know how to go about publishing a newsletter? Do you take full advantage of your RAMdisk? Have you yet to install an IBM mouse to work with your bridgeboard? Do you know there's an alternative to high-cost word processors? Do you still struggle through your directories?

Or if you're a programmer or technical type, do you understand how to add 512K RAM to your 1MB A500 for a cost of only $30? Or how to program the Amiga's GUI in C? Would you like the instructions for building your own variable rapid-fire joystick or a 246-grayscale SCSI interface for your Amiga? Do you use easy routines for performing floppy access without the aid of the operating system? How much do you really understand about ray tracing?

The answers to these questions and others can be found in *AMAZING COMPUTING* and *AC's TECH*.

SUBSCRIBE!

YES! *The "Amazing" AC publications give me **3 GREAT** reasons to save!*
Please begin the subscription(s) indicated below immediately!

Name____________________

Address ____________________

City__________ State ____ ZIP______

Charge my Visa MC #________________

Expiration Date ______ Signature ____________

Please circle to indicate this is a **New Subscription** or a **Renewal**

MasterCard® VISA® DISCOVER

Call now and use your Visa, Master Card, or Discover or fill out and send in this order form!

1 year of **AC**	12 big issues of Amazing Computing! Save over 43% off the cover price!	**US $27.00** ☐ **Canada/Mexico $34.00** ☐ **Foreign Surface $44.00** ☐
1-year SuperSub	AC + AC's GUIDE—14 issues total! Save more than 45% off the cover prices!	**US $37.00** ☐ **Canada/Mexico $54.00** ☐ **Foreign Surface $64.00** ☐
1 year of AC's TECH	4 big issues of the FIRST Amiga technical magazine with Disk!	**US $43.95** ☐ **Canada/Mexico $47.95** ☐ **Foreign Surface $51.95** ☐

Please call for all other Canada/Mexico/foreign Air Mail rates.
Check or money order payments must be in US funds drawn on a US bank; subject to applicable sales tax.

ORDER!

NAME____________________

ADDRESS____________________

CITY__________STATE____ZIP______

CHARGE MY: __VISA __M/C #________________

EXPIRATION DATE________ SIGNATURE____________

MasterCard® VISA® DISCOVER

Amazing Computing Back Issues: $5.00 each US, $6.00 each Canada and Mexico, $7.00 each Foreign Surface. Please list issue(s)____________
Amazing Computing Back Issue Volumes:
Volume 1-$15.00* Volume 2, 3, 4, 5, 6, 7, or 8—$20.00* each or any 12 issues for $20.00*
* All prices now include shipping & handling. * Foreign surface: $25. Air mail rates available.

AC's TECH/AMIGA **Single issues just $14.95! V1.1 (Premiere), V1.2, V1.3, V1.4, V2.1, V 2.2, V2.3, V2.4, V3.1, V3.2, V3.3, V3.4, V4.1 V4.2**
Volume One, Two, or Three (complete) or any four issues— $40.00!

Freely Distributable Software – Subscriber Special (yes, even the new ones!)

1 to 9 disks	**$6.00 each**
10 to 49 disks	**$5.00 each**
50 to 99 disks	**$4.00 each**
100 or more disks	**$3.00 each**

$7.00 each for non subscribers **(three disk minimum on all foreign orders)**

Amazing on Disk:

AC#1 ...Source & Listings V3.8 & V3.9	AC#2 ...Source & Listings V4.3 & V4.4
AC#3 ...Source & Listings V4.5 & V4.6	AC#4 ...Source & Listings V4.7 & V4.8
AC#5 ...Source & Listings V4.9	AC#6 ...Source & Listings V4.10 & V4.11
AC#7 ...Source & Listings V4.12 & V5.1	AC#8 ...Source & Listings V5.2 & 5.3
AC#9 ...Source & Listings V5.4 & V5.5	AC#10 ...Source & Listings V5.6 & 5.7
AC#11 ...Source & Listings V5.8, 5.9 & 5.10	AC#12 ...Source & Listings V5.11, 5.12 & 6.1
AC#13 ...Source & Listings V6.2 & 6.3	AC#14 ...Source & Listings V6.4, & 6.5
AC#15 ...Source & Listings V6.6, 6.7, 6.8, & 6.9	

Back Issues: $______

AC's TECH: $______

PDS Disks: $______

Total: $_____
(subject to applicable sales tax)

Please list your Freely Redistributable Software selections below:

AC Disks ______________
(numbers 1 through 15)
AMICUS ______________
(numbers 1 through 26)
Fred Fish Disks ______________
(numbers 1 through 1000)

Complete Today, or telephone 1-800-345-3360 now!

You may FAX your order to 1-508-675-6002

Please allow 4 to 6 weeks for delivery of subscriptions in US.

(Domestic and Foreign air mail rates available on request)

Check or money order payments must be in US funds drawn on a US bank; subject to applicable sales tax.

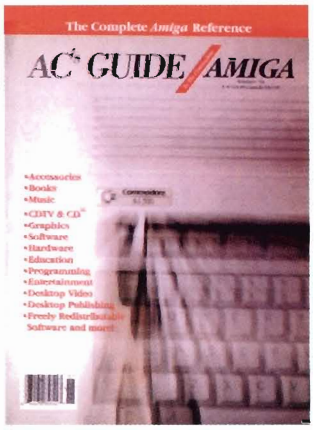
Maintain your edge...

AC's GUIDE gives you the latest in Amiga products and availability; now subscribe to the magazines that will keep you in the know.

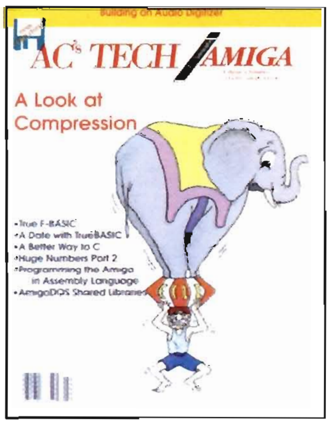
Subscribe to the best resources available for the *AMIGA*

Amazing Computing

AC's GUIDE



AC's TECH



Amazing Computing, the first Amiga monthly magazine, remains the first in new product announcements, unbiased reviews, and in-depth reporting. *AC*'s unique columns like *Roomers* and *Bug Bytes*, step-by-step programming articles, and entertaining tutorials have made it the magazine of choice with devoted Amiga fans. With *AC* you remain on the cutting edge of Amiga product development.

AC's GUIDE remains the world's best resource for Amiga product information. A compilation of new product announcements from *AC* and exhaustive research, *AC's GUIDE* is a constantly updated reference to the ever-changing Amiga market.

With an **AC SuperSub**, you will receive 12 issues of *Amazing Computing* and two issues of *AC's GUIDE* at a tremendous savings.

AC's TECH was the first disk-based technical magazine for the Amiga. This quarterly collection of programs, techniques, and developer issues has been created for Amiga owners who want to do more with their Amigas. If you want to expand your Amiga knowledge beyond the ordinary, then *AC's TECH* is a must.

Complete your *Amazing Computing* library and FRS collection

Mail or FAX (508-675-6002) the enclosed order form or call toll-free in U.S. or Canada, 800-345-3360.
Foreign orders please call 508-678-4200.

Special Offer for AC Readers!

AMOS (US), AMOS Compiler, and AMOS 3D

all three for only $99.99*

Bring your Amiga to *Life!*

AMOS - The Creator is like nothing you've ever seen before on the Amiga. If you want to harness the hidden power of your Amiga, then AMOS is for you!
AMOS Basic is a sophisticated development language with more than 500 different commands to produce the results you want with the minimum of effort. This special version of AMOS has been created to perfectly meet the needs of American Amiga owners. It includes clearer and brighter graphics than ever before, and a specially adapted screen size (NTSC).

"Whether you are a budding Amiga programmer who wants to create fancy graphics without weeks of typing, or a seasoned veteran who wants to build a graphic user interface with the minimum of fuss and link with C routines, AMOS is ideal for you." *Amazing Computing, June 1992*

HERE ARE JUST SOME OF THE ▶ ▶ THINGS YOU CAN DO ▶

- Define and animate hardware and software sprites (bobs) with lightning speed.
- Display up to eight screens on your TV at once - each with its own color palette and resolution (including HAM, interlace, half-brite and dual playfield modes).
- Scroll a screen with ease. Create multi-level parallax scrolling by overlapping different screens - perfect for scrolling shoot-em-ups.
- Use the unique AMOS Animation Language to create complex animation sequences for sprites, bobs or screens which work on interrupt.
- Play Soundtracker, Sonix or GMC (Games Music Creator) tunes or IFF samples on interrupt to bring your programs vividly to life.
- Use commands like RAINBOW and COPPER MOVE to create fabulous color bars like the very best demos.
- Transfer STOS programs to your Amiga and quickly get them working like the original.
- Use AMOS on any Amiga from an A500 with a single drive to the very latest model with hard disk.

WHAT YOU GET!

AMOS (US)—AMOS BASIC, sprite editor, Magic Forest and Amosteroids arcade games, Castle AMOS graphical adventure, Number Leap educational game, 400-page manual with more than 80 example programs on disk, sample tunes, sprite files, and registration card.
AMOS Compiler—AMOS Compiler, AMOS language updater, AMOS Assembler, eight demonstration programs which show off the power of the compiler, and a comprehensive, easy-to-use manual to develop lightning fast software.
AMOS 3D—Object Modeler, 30 new AMOS commands, and more. AMOS 3D allows you to create 3D animations as fast as 16 to 25 frames per second. You can display up to 20 objects at once, mix 3D with other AMOS features such as sprites, bobs, plus backgrounds, and more.

Limited Time Offer for AC readers only!

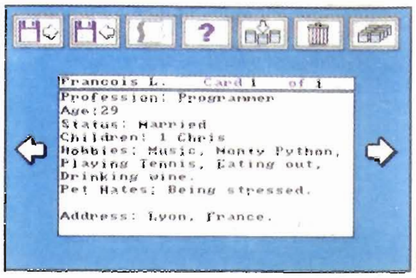
Get all three AMOS packages at one great price. Order today by sending your name, address (physical address please—all orders will be shipped by UPS), and $99.99 (*plus $10.00 for Shipping and handling) to: AMOS Special, PiM Publications, Inc., P.O. Box 2140, Fall River, MA 02722-2140 or use your VISA, MasterCard, or Discover and fax 1-508-675-6002 or call toll free in the US or Canada:

1-800-345-3360

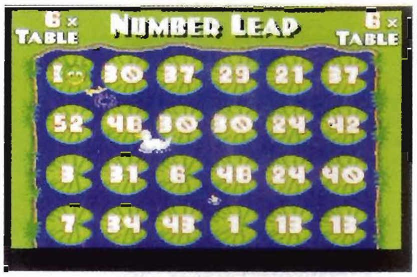
europress SOFTWARE

Please allow 4 to 6 weeks for delivery.
AMOS written by François Lionet.
©1992 Mandarin/Jawx
Country of Origin: UK

Use the sophisticated editor to design your creations


Francois L. Card 1 of 1
Profession: Programmer
Age: 29
Status: Married
Children: 1 Chris
Hobbies: Music, Monty Python, Playing Tennis, Eating out, Drinking wine.
Pet Hates: Being stressed.
Address: Lyon, France.

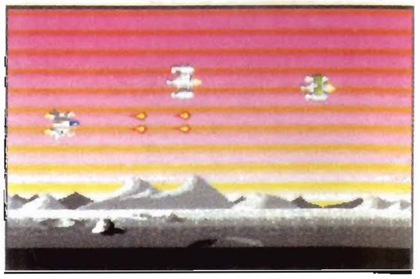
Create serious software like Dataflex



Produce educational programs with ease

Play Magic Forest and see just what AMOS can do!

Design sprites using the powerful Sprite Editor


Create breathtaking graphical effects as never before